UNIT 2: 2D Transformations, Viewing & Clipping

Introduction of Transformations

- Computer Graphics provide the facility of viewing object from different angles. The architect can study building from different angles i.e.
  - Front Evaluation
  - Side elevation
  - Top plan
- A Cartographer can change the size of charts and topographical maps. So if graphics images are coded as numbers, the numbers can be stored in memory.
- These numbers are modified by mathematical operations called as Transformation.

- The purpose of using computers for drawing is to provide facility to user to view the object from different angles, enlarging or reducing the scale or shape of object called as Transformation.
- Two essential aspects of transformation are given below:
  - Each transformation is a single entity. It can be denoted by a unique name or symbol.
  - It is possible to combine two transformations, after connecting a single transformation is obtained, e.g., A is a transformation for translation. The B transformation performs scaling. The combination of two is C=AB. So C is obtained by concatenation property.

- There are two complementary points of view for describing object transformation.
  - **Geometric Transformation:** The object itself is transformed relative to the coordinate system or background. The mathematical statement of this viewpoint is defined by geometric transformations applied to each point of3 the object.
  - **Coordinate Transformation:** The object is held stationary while the coordinate system is transformed relative to the object. This effect is attained through the application of coordinate transformations.

- Types of Transformations:
  - Translation
  - Scaling
  - Rotating
  - Reflection
  - Shearing

1. Translation

- We perform a translation on a single coordinate point by adding offsets to its coordinates so as to generate a new coordinate position.

- It is the straight line movement of an object from one position to another is called Translation. Here the object is positioned from one coordinate location to another.

- To translate a 2D position, we add translation distances tx and ty (that pair is also called translation vector or shift vector) to the original coordinates (x,y) to obtain the new coordinate position (x',y').

$$x' = x + t_x, \qquad y' = y + t_y$$

$$\mathbf{P} = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}, \qquad \mathbf{P}' = \begin{bmatrix} x_1' \\ x_2' \end{bmatrix}, \qquad \mathbf{T} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

$$\mathbf{P}' = \mathbf{P} + \mathbf{T}$$

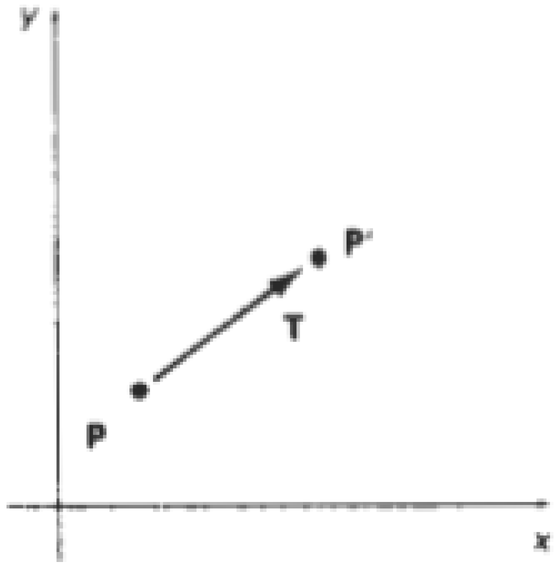
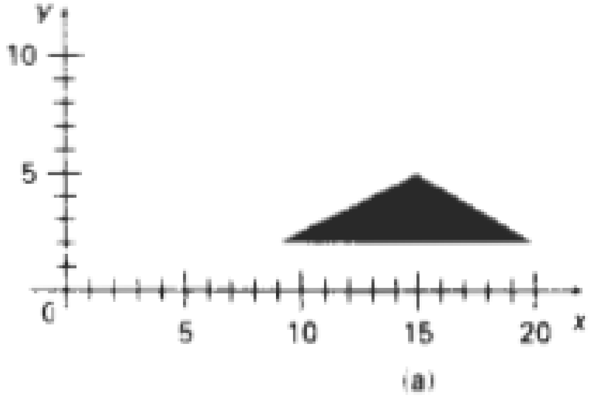


Figure 5-1
Translating a point from position **P** to position **P'** with translation vector **T**.



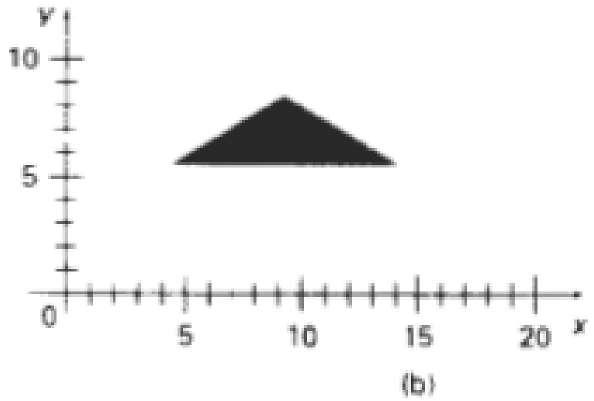


Figure 5-2
Moving a polygon from position (a) to position (b) with the translation vector (-5.50, 3.75).

2. Scaling

- To alter the size of an object, we apply a scaling transformation.
- A simple two-dimensional scaling operation is performed by multiplying object position (x,y) by scaling factors sx and sy to produce the transformed coordinates (x',y').

$$x' = x \cdot sx, \qquad y' = y \cdot sy$$

- Scaling factor sx scales an object in the x direction, while sy scales in the y direction.
- The basic two-dimensional scaling equation can also be written in the following matrix form:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

$$\mathbf{P}' = \mathbf{S} \cdot \mathbf{P}$$

- Any positive values can be assigned to the scaling factors sx and sy.
- Values less than 1 reduce the size of objects; values greater than 1 produce enlargements.
- Specifying a value of 1 for both sx and sy leaves the size of objects unchanged.

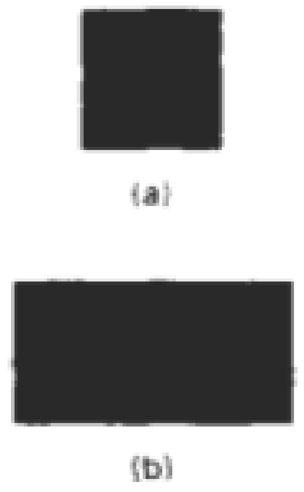


Figure 5-6
Turning a square (a) into a rectangle (b) with scaling factors $s_x = 2$ and $s_y = 1$.

- When sx and sy are assigned the same value, a uniform scaling is produced, which maintains relative object proportions.
- Unequal values for sx and sy result in a differential scaling that is often used in design applications, where pictures are constructed from a few basic shapes that can be adjusted by scaling and positioning transformation.

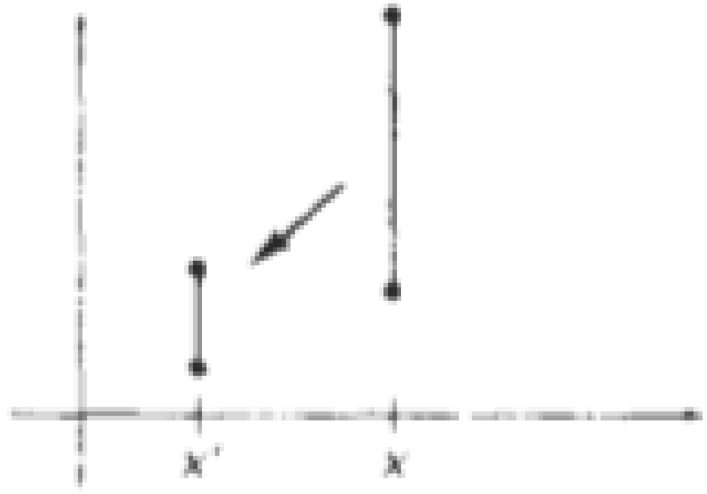
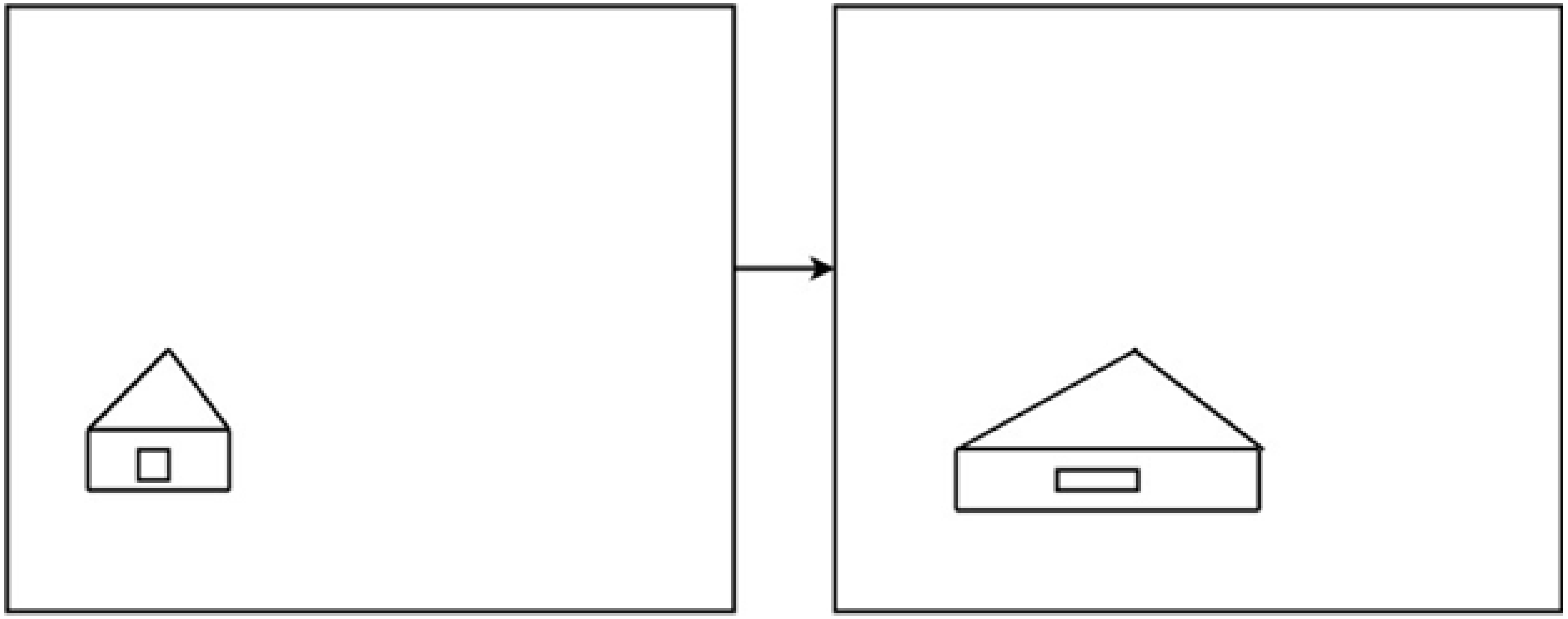


Figure 5-7
A line scaled with Eq 5-12 using $s_x = s_y = 0.5$ is reduced in size and moved closer to the coordinate origin.


Original Object
Object after scaling in X direction

3. Rotation

- We generate a rotation transformation of an object by specifying a rotation axis and a rotation angle.
- All the points of the object are then transformed to new positions by rotating the points through the specified angle about the rotation axis.
- A two-dimensional rotation of an object is obtained by repositioning the object along a circular path in the xy plane.

- Parameters for the two-dimensional rotation are the rotation angle θ and a position (xr,yr), called the rotation point or pivot point, about which the object is to be rotated .
- The pivot point is the intersection position of the rotation axis with the xy plane.
- A positive value for the angle θ defines a counterclockwise rotation about the pivot point and negative value rotates objects in the clockwise direction.

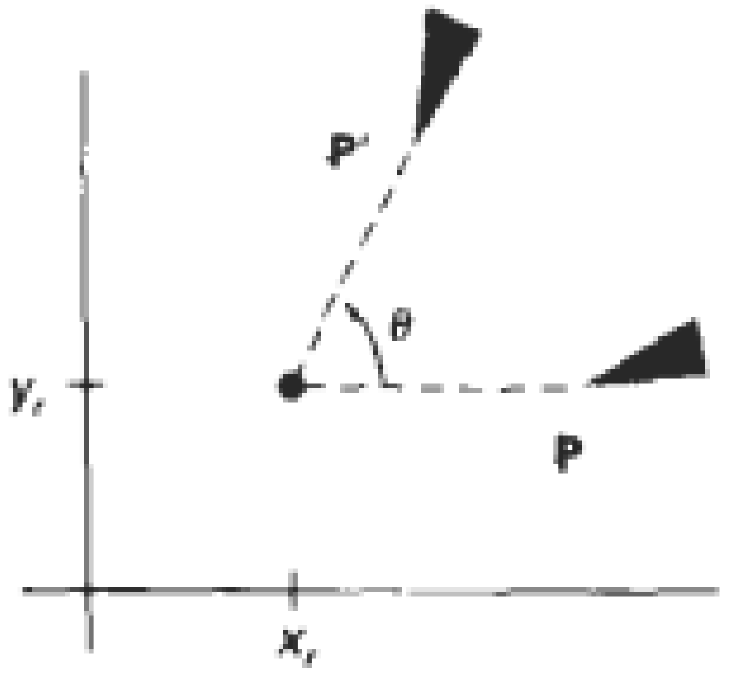


Figure 5-3
Rotation of an object through angle θ about the pivot point (x_r, y_r).

- In the figure, r is the constant distance of the point from the origin, angle ϕ is the original angle position of the point from the horizontal, and θ is the rotation angle.
- Using standard trigonometric identities we can express as:

$$x' = r\cos(\phi + \theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$$

$$y' = r\sin(\phi + \theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta$$

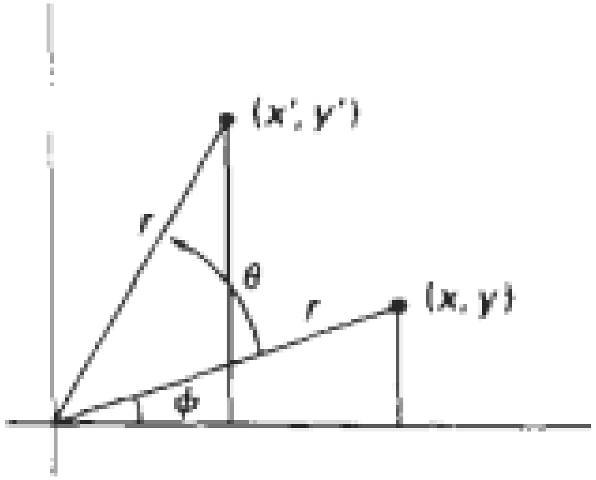


Figure 5-4
Rotation of a point from position (x, y) to position (x', y') through an angle θ relative to the coordinate origin. The original angular displacement of the point from the x axis is ϕ.

$$x = r\cos\phi, \qquad y = r\sin\phi$$

$$x' = x\cos\theta - y\sin\theta$$
$$y' = x\sin\theta + y\cos\theta$$

$$\mathbf{P}' = \mathbf{R}\cdot\mathbf{P}$$

$$\mathbf{R} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

Homogeneous Coordinates System

- Many graphics applications involve sequences of geometric transformations.
- In design and picture construction applications, we perform translations, rotations, and scalings to fit the picture components into their proper positions.

$$\mathbf{P}' = \mathbf{M}_1 \cdot \mathbf{P} + \mathbf{M}_2$$

- We can combine the multiplicative and translational terms for 2D geometric transformations into a single matrix representation by expanding the 2x2 matrix representations to 3x3 matrices.
- To express any 2D transformation as a matrix multiplication, we represent each Cartesian coordinate position (x, y) with the homogeneous coordinate triple (xh, yh, h), where:

$$x = \frac{x_h}{h}, \qquad y = \frac{y_h}{h}$$

- For 2D geometric transformations, we can choose the homogeneous parameter h to be any nonzero value.
- Thus, there is an infinite number of equivalent homogeneous representations for each coordinate point (x, y).
- A convenient choice is simple to set h = 1. Each 2D position is then represented with homogeneous coordinates (x, y, 1).
- The term homogeneous coordinates is used in mathematics to refer to the effect of this representation in Cartesian equations.

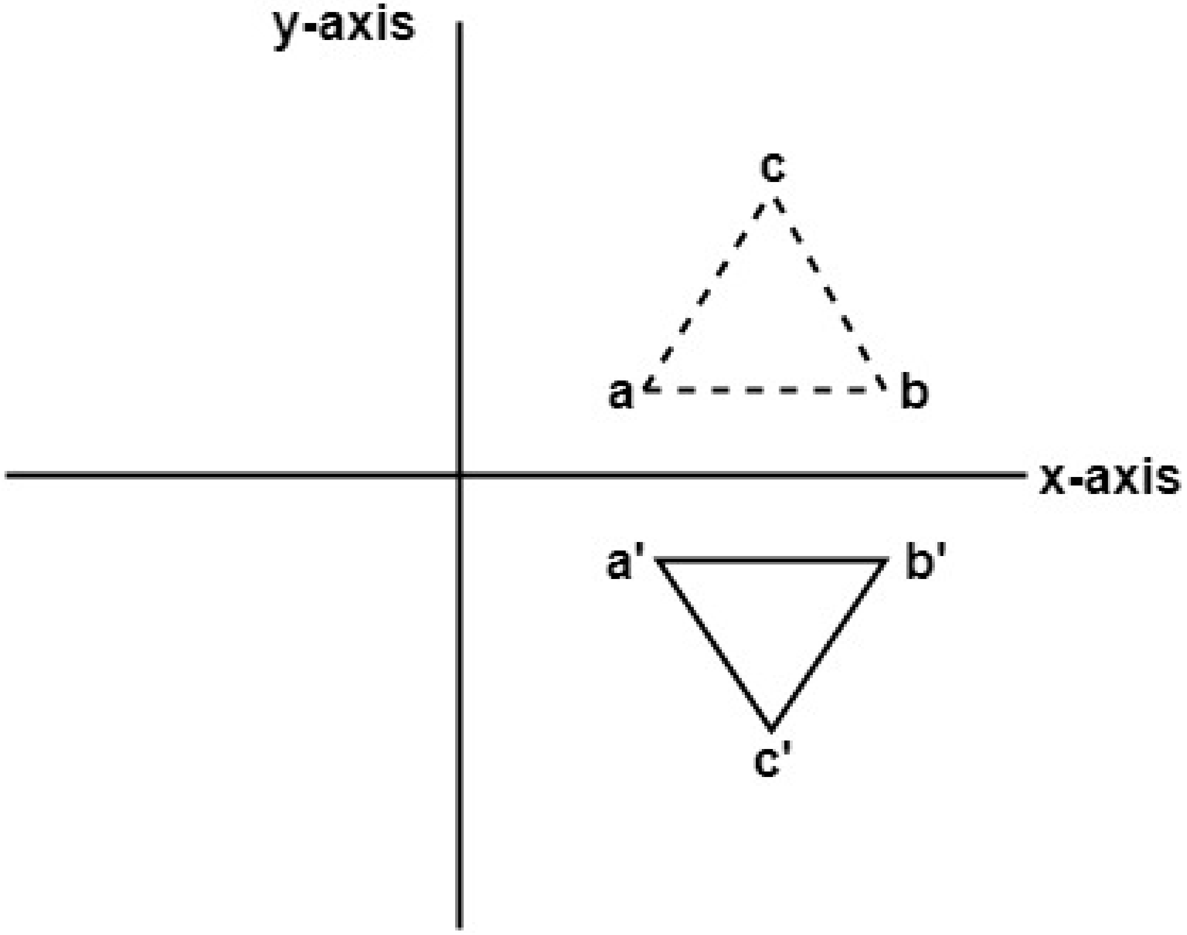
$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \mathbf{P}' = \mathbf{T}(t_x, t_y) \cdot \mathbf{P}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \mathbf{P}' = \mathbf{R}(\theta) \cdot \mathbf{P}$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \qquad \mathbf{P}' = \mathbf{S}(s_x, s_y) \cdot \mathbf{P}$$

Other transformations: Reflection

- It is a transformation which produces a mirror image of an object. The mirror image can be either about x-axis or y-axis. The object is rotated by180°.
- When the reflection axis is a line in the xy plane, the rotation path about this axis is in a plane perpendicular to the xy plane.
- For reflection axes that are perpendicular to the xy plane, the rotation path is in the xy plane.


y-axis
c
a
b
x-axis
a'
b'
c'

- Reflection about the line y = 0, the x axis, is accomplished with the transformation matrix:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Reflection about the y axis flips x coordinates while keeping y coordinates the same. The matrix for this transformation is:

$$\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

4. Reflection…

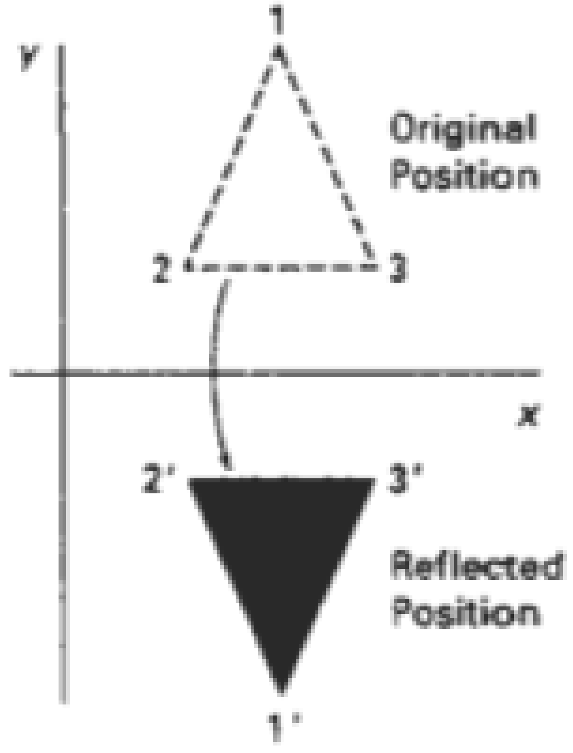


Figure 5-16
Reflection of an object about the *x* axis.

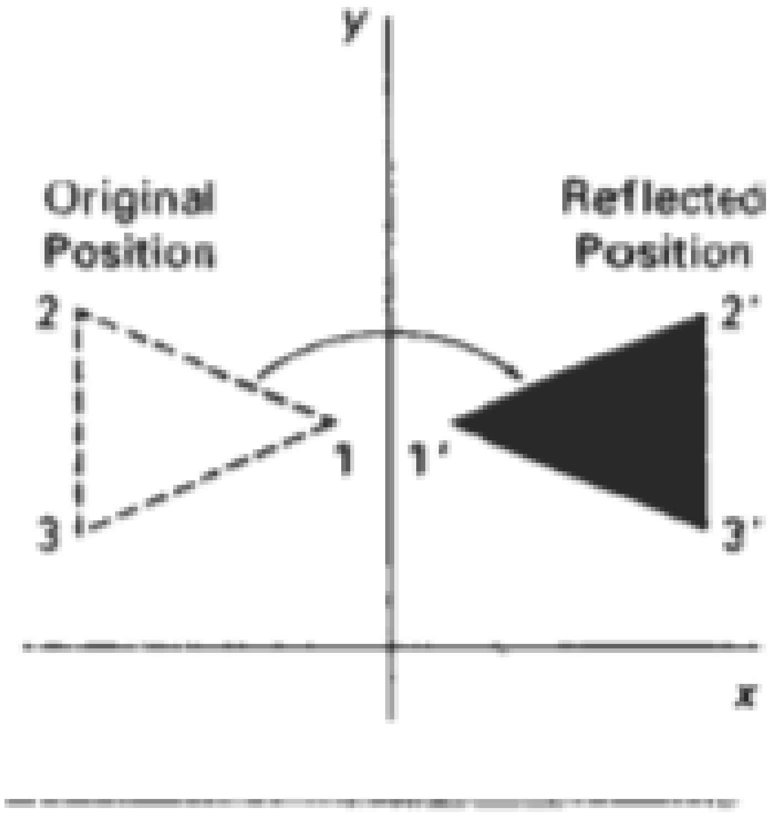


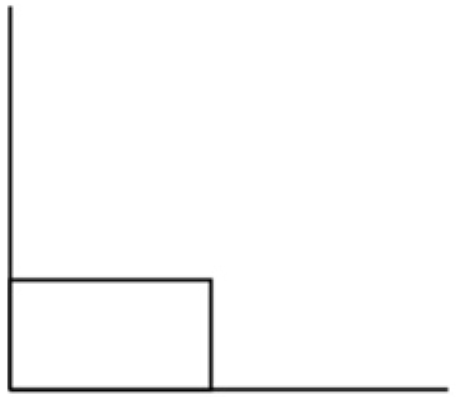
Figure 5-17
Reflection of an object about the *y* axis.

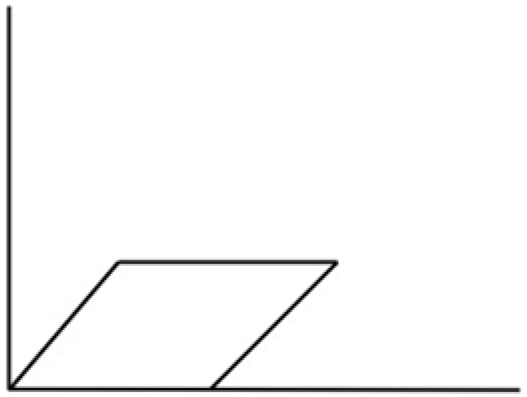
5. Shearing

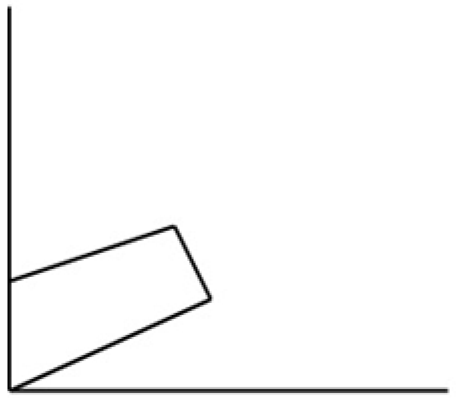
- A transformation that distorts the shape of an object such that the transformed shape appears as if the object were composed in the internal layers that had been caused to slide over each other is called a shear.

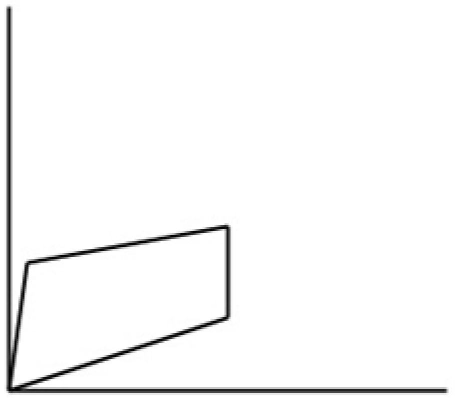
- Two common shearing transformations are those that shift coordinate x values and those that shift y values.

5. Shearing…


Original Object


Shear in X direction


Shear in Y direction


Shear in both directions

- An x-direction shear relative to the x axis is produced with the transformation matrix:

$$\begin{bmatrix} 1 & sh_x & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Which transforms coordinate position as:

$$x' = x + sh_x \cdot y, \qquad y' = y$$

- We can generated x-direction shears relative to other reference lines with: $\begin{bmatrix} 1 & sh_x & -sh_x \cdot y_{ref} \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

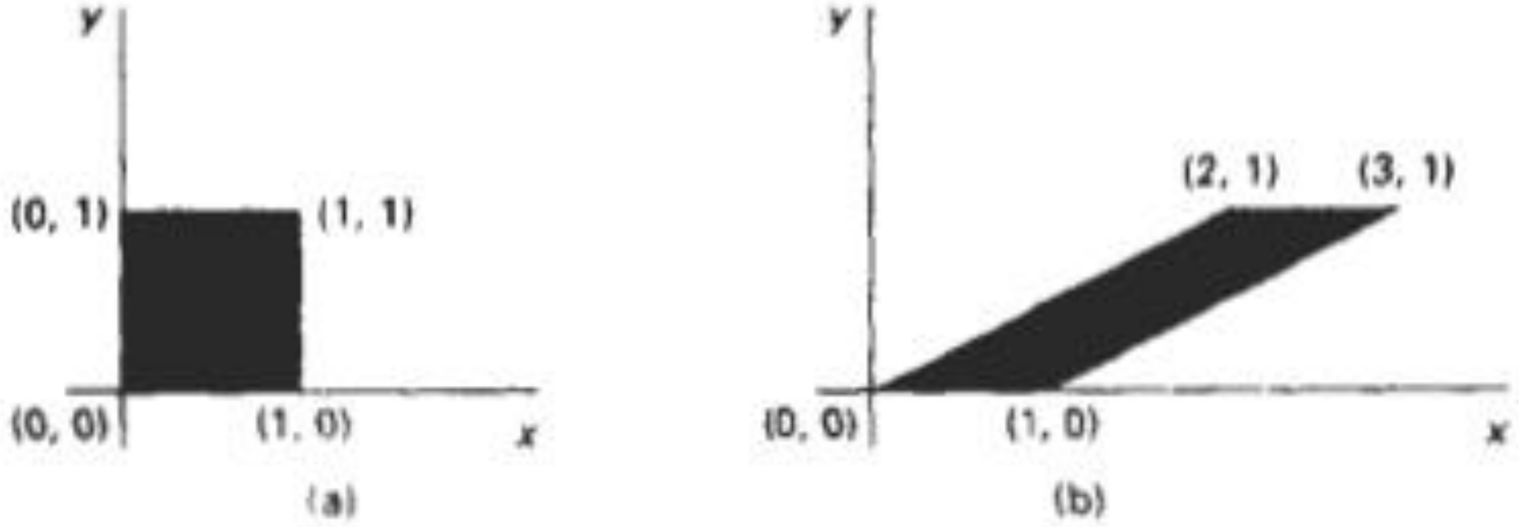


Figure 5-23

A unit square (a) is converted to a parallelogram (b) using the x-direction shear matrix 5-53 with $sh_x = 2$.

Problems: 2D Geometric Transformations

1. Translate a polygon with coordinates A(2,5), B(7,10) and C(10,2) by 3 units in x direction and 4 units in y direction.
2. A point (4,3) is rotated counterclockwise by an angle 45 degree. Find the rotation matrix and the resultant point.
3. Scale the polygon with coordinate A(2,5), B(7,10) and C(10,2) by 2 units in x direction and 2 units in y direction.

Problems: 2D Homogeneous coordinate

- Give a 3x3 homogeneous coordinate transformation matrix for each of the following translations:
  - Shift the image to the right 3 units
  - Shift the image up 2 units
  - Move the image down 0.5 unit and right 1 unit
  - Move the image down 0.66 unit and left 4 units

Inverse Transformations

- For translation, we obtain the inverse matrix by negating the translation distances.
- Thus, if we have 2D translation distance tx and ty, the inverse translation matrix:

$$\mathrm{T}^{-1} = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix}$$

- This produces a translation in the opposite direction, and the product of a translation matrix and its inverse produces the identity matrix.

- An inverse rotation is accomplished by replacing the rotation angle by its negative.

$$R^{-1} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Negative values for rotation angles generate rotations in a clockwise direction, so the identity matrix is produced when any rotation matrix is multiplied by its inverse.
- Because only the sine function is affected by the change in sign of the rotation angle, the inverse matrix can also be obtained by interchanging rows and columns.

- Now, we form the inverse matrix for any scaling transformation by replacing the scaling parameters with their reciprocals.
- For 2D scaling with parameters sx and sy applied to the coordinate origin, the inverse transformation matrix is:

$$\mathbf{S}^{-1} = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- The inverse matrix generates an opposite scaling transformation, so the multiplication of any scaling matrix with its inverse produces the identity matrix.

2D Composite Transformations

- With the matrix representations, we can set up a matrix for any sequence of transformations as a composite transformations matrix by calculating the matrix product of the individual transformations.
- Forming products of transformation matrices is often referred to as a concatenation, or composition, of matrices.
- Because a coordinate position is represented with a homogeneous column matrix, we must pre-multiply the column matrix by the matrices representing any transformation sequence.

- Also, because many positions in a scene are typically transformed by the same sequence, it is more efficient to first multiply the transformation matrices to form a single composite matrix.

- Thus, if we want to apply two transformations to point position P, the transformed location would be calculated as: P’ = M2 . M1 . P

  = M . P

Composite 2D Translations

- If two successive translation vectors (t1x , t1y) and (t2x , t2y) are applied to a two-dimensional coordinate position P, the final transformed location P' is calculated as:

$$\begin{aligned} \mathbf{P}' &= \mathbf{T}(t_{2x}, t_{2y}) \cdot \{\mathbf{T}(t_{1x}, t_{1y}) \cdot \mathbf{P}\} \\ &= \{\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y})\} \cdot \mathbf{P} \end{aligned}$$

- We can verify this result by calculating the matrix product for the two associative groupings.

- Also, the composite transformation matrix for this sequence of translations is:

$$\begin{bmatrix} 1 & 0 & t_{2x} \\ 0 & 1 & t_{2y} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & t_{1x} \\ 0 & 1 & t_{1y} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & t_{1x} + t_{2x} \\ 0 & 1 & t_{1y} + t_{2y} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{T}(t_{2x}, t_{2y}) \cdot \mathbf{T}(t_{1x}, t_{1y}) = \mathbf{T}(t_{1x} + t_{2x}, t_{1y} + t_{2y})$$

- Which demonstrates that two successive translations are additive.

Composite 2D Rotations

- Two successive rotations applied to a point P produce the transformed position:

$$\begin{aligned}\mathbf{P}' &= \mathbf{R}(\theta_2) \cdot \{\mathbf{R}(\theta_1) \cdot \mathbf{P}\} \\ &= \{\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1)\} \cdot \mathbf{P}\end{aligned}$$

- By multiplying the two rotation matrices, we can verify that two successive rotations are additive:

$$\mathbf{R}(\theta_2) \cdot \mathbf{R}(\theta_1) = \mathbf{R}(\theta_1 + \theta_2)$$

- so that the final rotated coordinates of a point can be calculated with the composite rotation matrix as:

$$\mathbf{P}' = \mathbf{R}(\theta_1 + \theta_2) \cdot \mathbf{P}$$

Composite 2D Scalings

- Concatenating transformation matrices for two successive scaling operations in two dimensions produces the following composite scaling matrix:

$$\begin{bmatrix} s_{2x} & 0 & 0 \\ 0 & s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_{1x} & 0 & 0 \\ 0 & s_{1y} & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_{1x} \cdot s_{2x} & 0 & 0 \\ 0 & s_{1y} \cdot s_{2y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{S}(s_{2x}, s_{2y}) \cdot \mathbf{S}(s_{1x}, s_{1y}) = \mathbf{S}(s_{1x} \cdot s_{2x},\ \ s_{1y} \cdot s_{2y})$$

- The resulting matrix in this case indicates that successive scaling operations are multiplicative.

- That is, if we were to triple the size of an object twice in succession, the final size would be nine times that of the original.

General 2D Pivot-Point Rotation

- When a graphics package provides only a rotate function with respect to the coordinate origin, we can generate a two-dimensional rotation about any other pivot point (xr , yr) by performing the following sequence of translate-rotate-translate operations:

  1. **Translate** the object so that the pivot-point position is moved to the coordinate origin.

  2. **Rotate** the object about the coordinate origin.

  3. **Translate** the object so that the pivot point is returned to its original position.

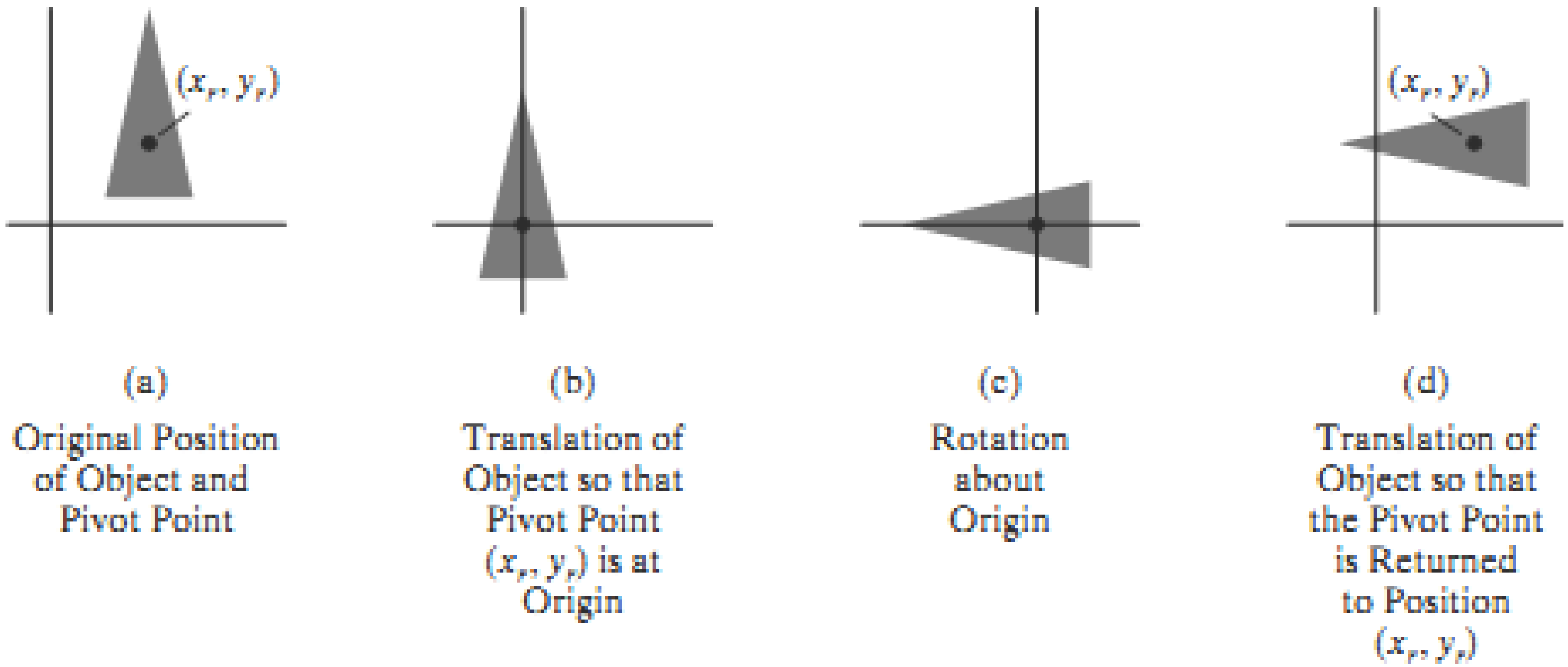

(x_r, y_r)
(x_r, y_r)
(a)
Original Position of Object and Pivot Point
(b)
Translation of Object so that Pivot Point (x_r, y_r) is at Origin
(c)
Rotation about Origin
(d)
Translation of Object so that the Pivot Point is Returned to Position (x_r, y_r)

FIGURE 9
A transformation sequence for rotating an object about a specified pivot point using the rotation matrix **R**(θ) of transformation 19.

$$\begin{bmatrix} 1 & 0 & x_r \\ 0 & 1 & y_r \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_r \\ 0 & 1 & -y_r \\ 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta) + y_r \sin\theta \\ \sin\theta & \cos\theta & y_r(1-\cos\theta) - x_r \sin\theta \\ 0 & 0 & 1 \end{bmatrix}$$

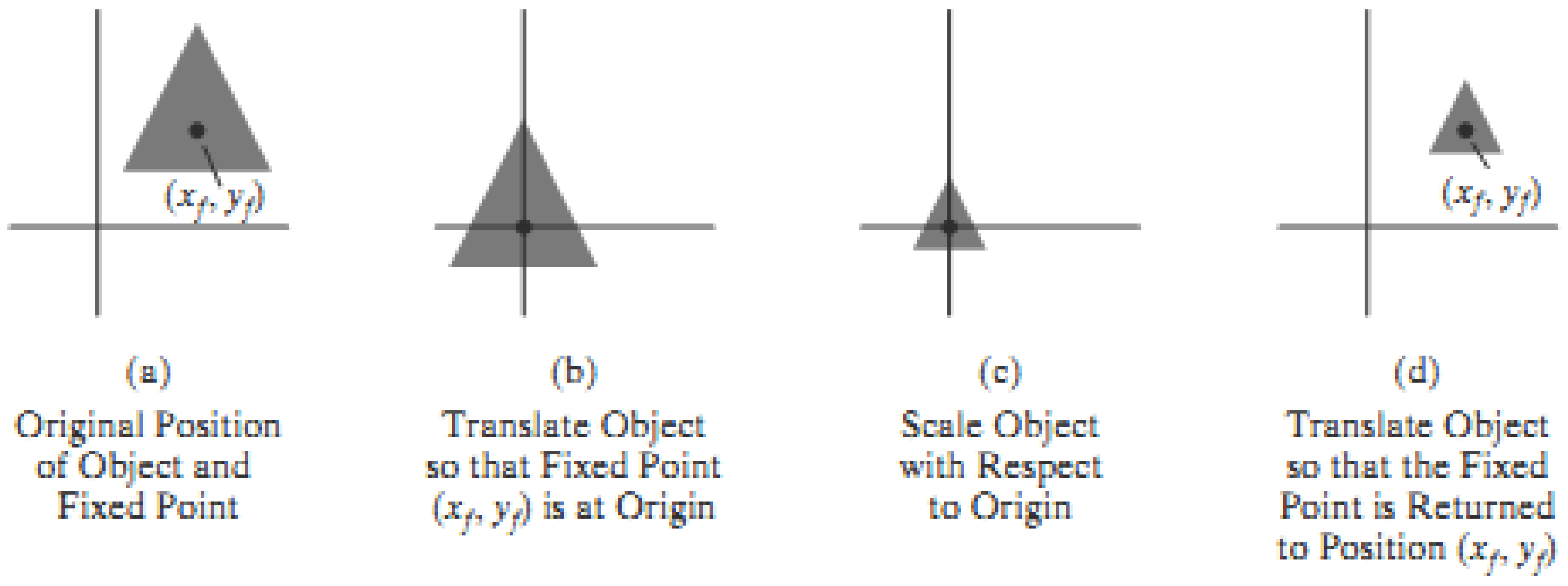
which can be expressed in the form:

$$\mathbf{T}(x_r, y_r) \cdot \mathbf{R}(\theta) \cdot \mathbf{T}(-x_r, -y_r) = \mathbf{R}(x_r, y_r, \theta)$$

General 2D Fixed-Point Scaling

- A transformation sequence to produce a two-dimensional scaling with respect to a selected fixed position (x f , y f), when we have a function that can scale relative to the coordinate origin only. This sequence is:

1. Translate the object so that the fixed point coincides with the coordinate origin.
2. Scale the object with respect to the coordinate origin.
3. Use the inverse of the translation in step (1) to return the object to its original position



(x_f, y_f)
(a)
Original Position of Object and Fixed Point
(b)
Translate Object so that Fixed Point (x_f, y_f) is at Origin
(c)
Scale Object with Respect to Origin
(x_f, y_f)
(d)
Translate Object so that the Fixed Point is Returned to Position (x_f, y_f)

FIGURE 10
A transformation sequence for scaling an object with respect to a specified fixed position using the scaling matrix $\mathbf{S}(s_x, s_y)$ of transformation 21.

- Concatenating the matrices for these three operations produces the required scaling matrix:

$$\begin{bmatrix} 1 & 0 & x_f \\ 0 & 1 & y_f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -x_f \\ 0 & 1 & -y_f \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & x_f(1-s_x) \\ 0 & s_y & y_f(1-s_y) \\ 0 & 0 & 1 \end{bmatrix}$$

or

$$\mathbf{T}(x_f, y_f) \cdot \mathbf{S}(s_x, s_y) \cdot \mathbf{T}(-x_f, -y_f) = \mathbf{S}(x_f, y_f, s_x, s_y)$$

- This transformation is generated automatically in systems that provide a scale function that accepts coordinates for the fixed point.

General 2D Scaling Directions

- Parameters sx and sy scale objects along the x and y directions.
- We can scale an object in other directions by rotating the object to align the desired scaling directions with the coordinate axes before applying the scaling transformation.
- Suppose we want to apply scaling factors with values specified by parameters s1 and s2 in the directions.

- To accomplish the scaling without changing the orientation of the object, we first perform a rotation so that the directions for s1 and s2 coincide with the x and y axes, respectively.

- Then the scaling transformation S(s1, s2) is applied, followed by an opposite rotation to return points to their original orientations.

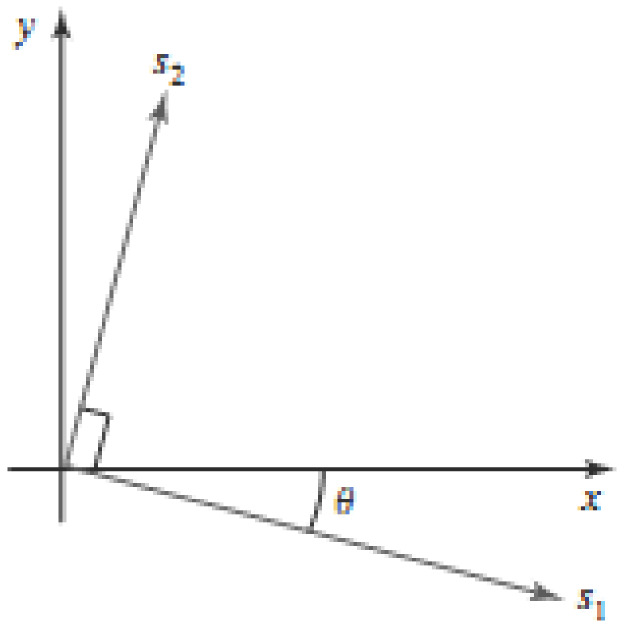


FIGURE 11
Scaling parameters s_1 and s_2 along orthogonal directions defined by the angular displacement θ.

- The composite matrix resulting from the product of these three transformations is

$$\mathbf{R}^{-1}(\theta)\cdot\mathbf{S}(s_1,s_2)\cdot\mathbf{R}(\theta)=\begin{bmatrix} s_1\cos^2\theta+s_2\sin^2\theta & (s_2-s_1)\cos\theta\sin\theta & 0\\ (s_2-s_1)\cos\theta\sin\theta & s_1\sin^2\theta+s_2\cos^2\theta & 0\\ 0 & 0 & 1\end{bmatrix}$$

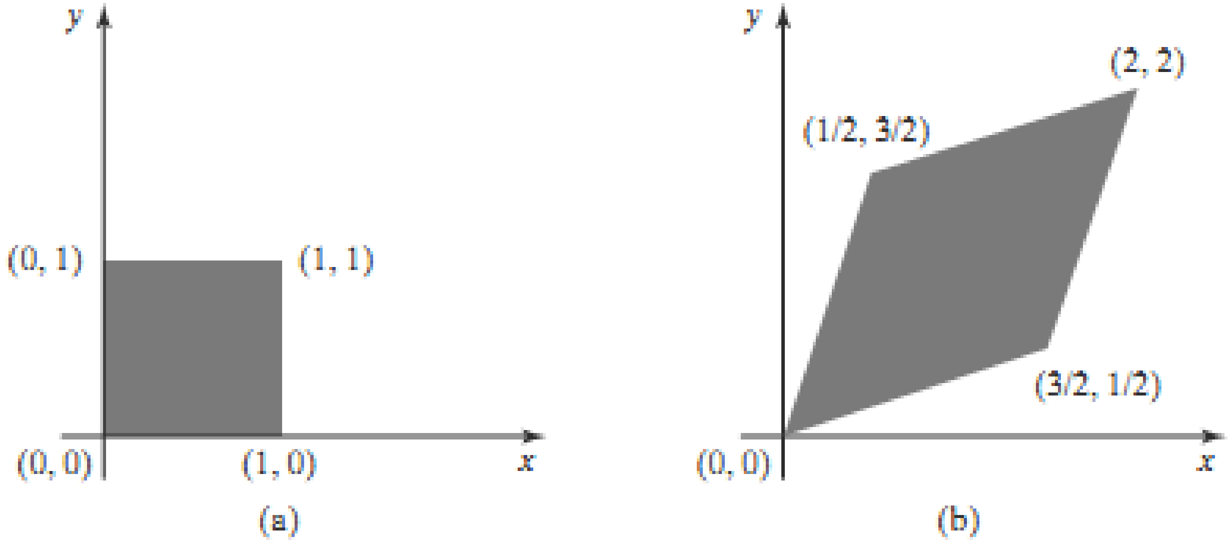


FIGURE 12
A square (a) is converted to a parallelogram (b) using the composite transformation matrix 39, with $s_1 = 1$, $s_2 = 2$, and $\theta = 45°$.

Problems

- Show how shear transformation may be expressed in terms of rotation and scaling.
- Apply the shearing transformation to square with A(0,0), B(1,0), C(1,1) and D(0,1) as given below:
  - Shear parameter value of 0.5 relative to the line yref = -1
  - Shear parameter value of 0.5 relative to the line xref = -1

2-Dimensional Viewing and Clipping

- A graphics package allows a user to specify which part of a defined picture is to be displayed and where that part is to be placed on the display device.
- Any convenient Cartesian coordinate system, referred to as the world-coordinate reference frame, can be used to define the picture.
- For a 2D picture, a view is selected by specifying a region of the xy plane that contains the total picture or any part of it.
- The picture parts within the selected areas are then mapped onto specified areas of the device coordinates.
- When multiple view areas are selected, these areas can be placed in separate display locations, or some areas could be inserted into other, larger display areas.

1. The 2D Viewing Pipeline

- A section of a two-dimensional scene that is selected for display is called a **clipping window** because all parts of the scene outside the selected section are “clipped” off.
- The only part of the scene that shows up on the screen is what is inside the clipping window.
- Sometimes the clipping window is alluded to as the **world window** or the **viewing window**.

- Graphics packages allow us also to control the placement within the display window using another “window” called the viewport.
- Objects inside the clipping window are mapped to the viewport, and it is the viewport that is then positioned within the display window.
- The clipping window selects what we want to see; the viewport indicates where it is to be viewed on the output device.
- By changing the position of a viewport, we can view objects at different positions on the display area of an output device.
- Multiple viewports can be used to display different sections of a scene at different screen positions.
- Usually, clipping windows and viewports are rectangles in standard position, with the rectangle edges parallel to the coordinate axes.

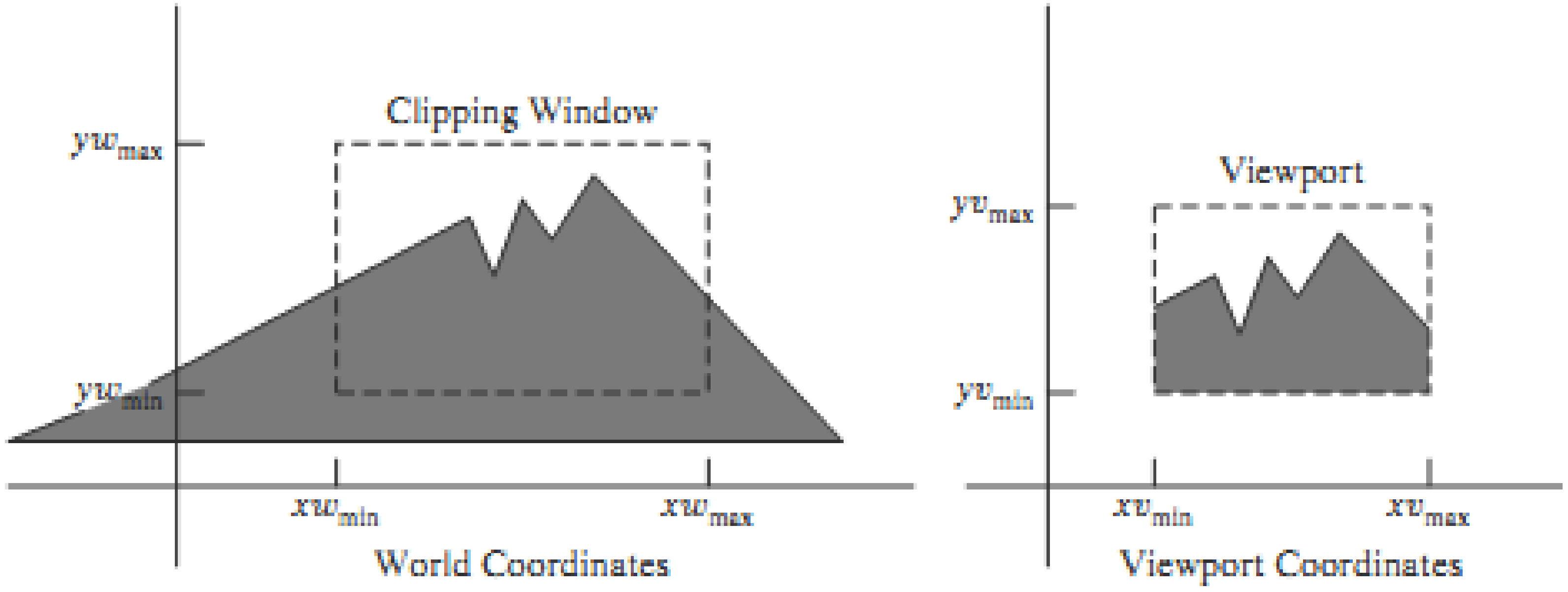


FIGURE 1
A clipping window and associated viewport, specified as rectangles aligned with the coordinate axes.

- The mapping of a two-dimensional, world-coordinate scene description to device coordinates is called a two-dimensional viewing transformation. Sometimes this transformation is simply referred to as the window-to-viewport transformation or the windowing transformation.

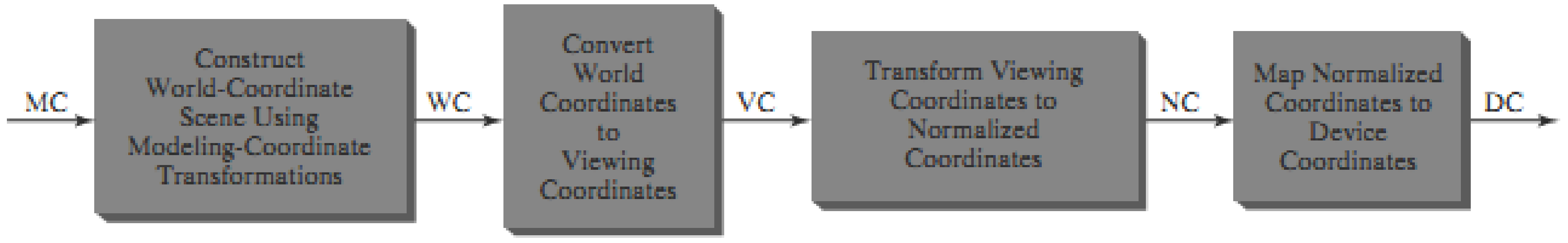

MC
Construct World-Coordinate Scene Using Modeling-Coordinate Transformations
WC
Convert World Coordinates to Viewing Coordinates
VC
Transform Viewing Coordinates to Normalized Coordinates
NC
Map Normalized Coordinates to Device Coordinates
DC

FIGURE 2
Two-dimensional viewing-transformation pipeline.

- Once a world-coordinate scene has been constructed, we could set up a separate two-dimensional, viewing-coordinate reference frame for specifying the clipping window.
- But the clipping window is often just defined in world coordinates, so viewing coordinates for two-dimensional applications are the same as world coordinates.
- Clipping is usually performed in normalized coordinates.
- This allows us to reduce computations by first concatenating the various transformation matrices.
- Clipping procedures are of fundamental importance in computer graphics.

2. The Clipping Window

- To achieve a particular viewing effect in an application program, we could design our own clipping window with any shape, size, and orientation we choose.
- The simplest window edges to clip against are straight lines that are parallel to the coordinate axes.
- Therefore, graphics packages commonly allow only rectangular clipping windows aligned with the x and y axes.
- Rectangular clipping windows in standard position are easily defined by giving the coordinates of two opposite corners of each rectangle.

Viewing-Coordinate Clipping Window

- A general approach to the two-dimensional viewing transformation is to set up a viewing-coordinate system within the world-coordinate frame.
- This viewing frame provides a reference for specifying a rectangular clipping window with any selected orientation and position.
- To obtain a view of the world-coordinate scene as determined by the clipping window of, we just need to transfer the scene description to viewing coordinates.

By changing the position of the viewport

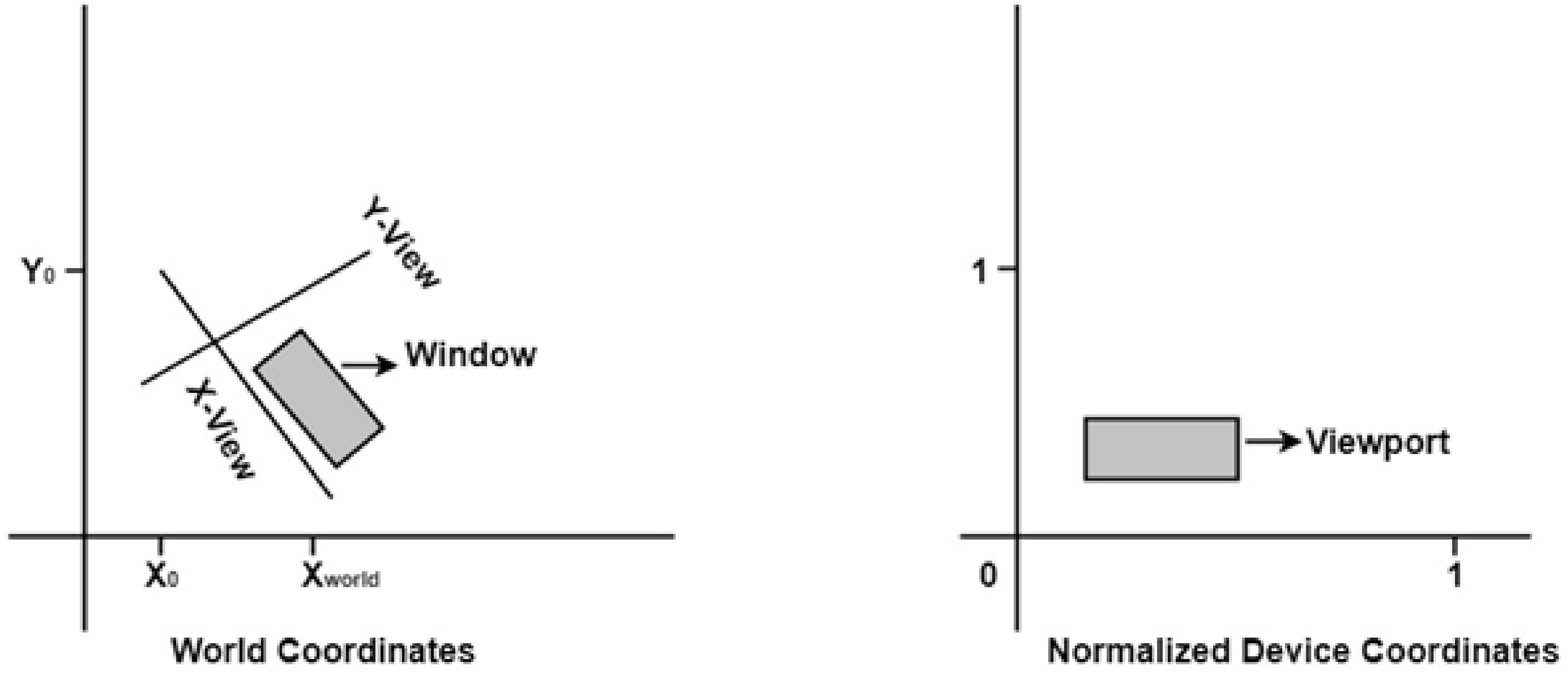


Fig:Setting up a rotated world window and corresponding normalized coordinate viewport.

- We choose an origin for a two-dimensional viewing-coordinate frame at some world position P0 = (x0, y0), and we can establish the orientation using a world vector V that defines the yview direction. Vector V is called the two-dimensional view up vector.

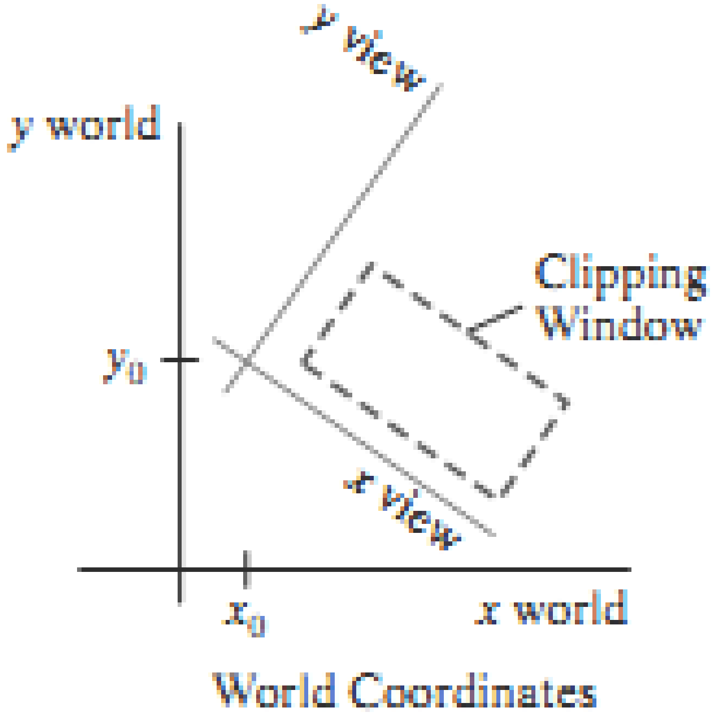


FIGURE 3
A rotated clipping window defined in viewing coordinates.

- Once we have established the parameters that defines the viewing-coordinate frame, we transform the scene description to the viewing system.
- The first step in the transformation sequence is to translate the viewing origin to the world origin.
- Next, we rotate the viewing system to align it with the world frame.
- Object positions in world coordinates are then converted to viewing coordinates with the composite two-dimensional transformation matrix:

$$\mathbf{M}_{WC,VC} = \mathbf{R} \cdot \mathbf{T}$$

World-Coordinate Clipping Window

- A routine for defining a standard, rectangular clipping window in world coordinates is typically provided in a graphics-programming library.
- We simply specify two world-coordinate positions, which are then assigned to the two opposite
- Once the clipping window has been established, the scene description is processed through the viewing routines to the output device.

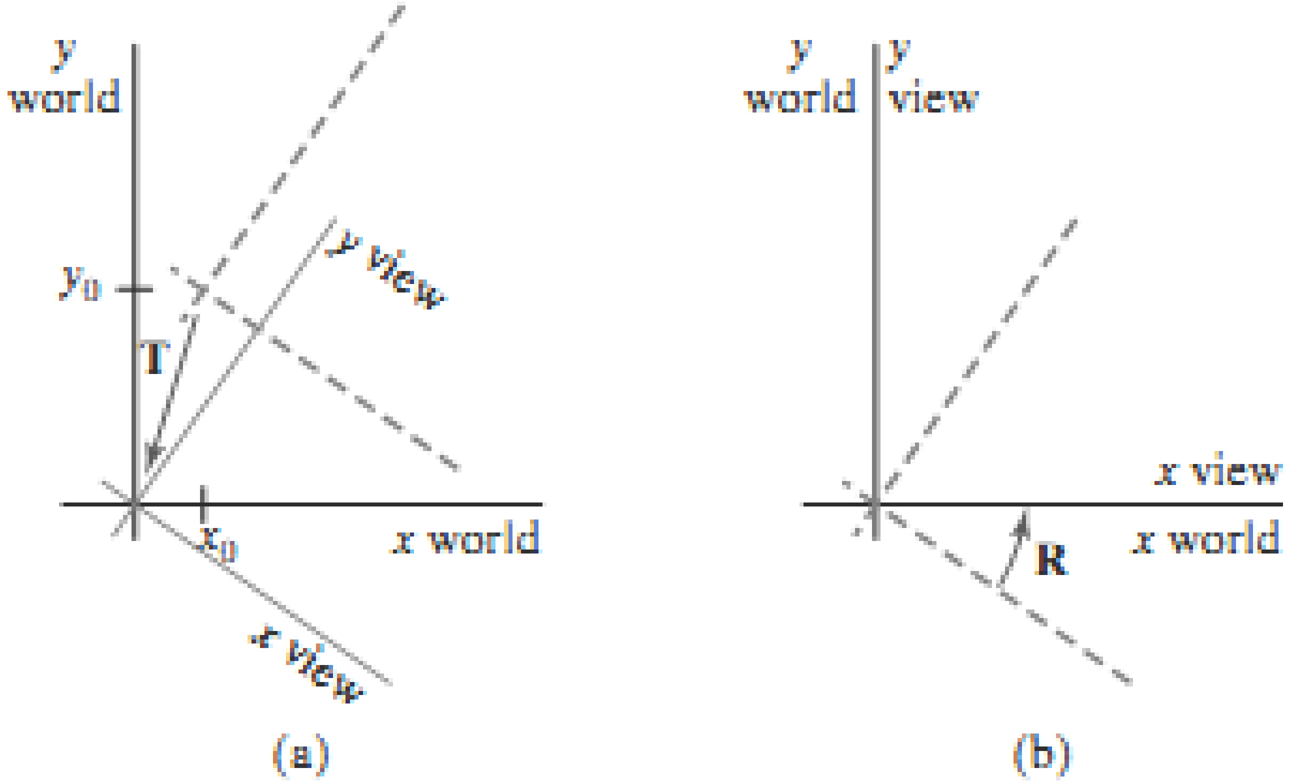


FIGURE 4

A viewing-coordinate frame is moved into coincidence with the world frame by (a) applying a translation matrix **T** to move the viewing origin to the world origin, then (b) applying a rotation matrix **R** to align the axes of the two systems.

- If we want to obtain a rotated view of a two-dimensional scene, as discussed in the previous section, we perform exactly the same steps as described there, but without considering a viewing frame of reference.

- Thus, we simply rotate (and possibly translate) objects to a desired position and set up the clipping window—all in world coordinates.

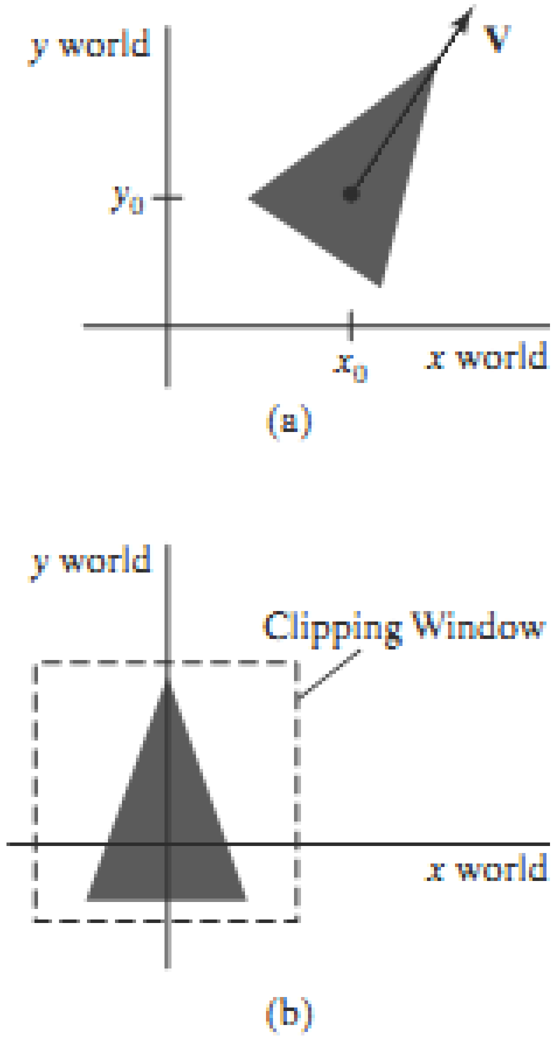


FIGURE 5
A triangle (a), with a selected reference point and orientation vector, is translated and rotated to position (b) within a clipping window.

By varying the size of the viewports

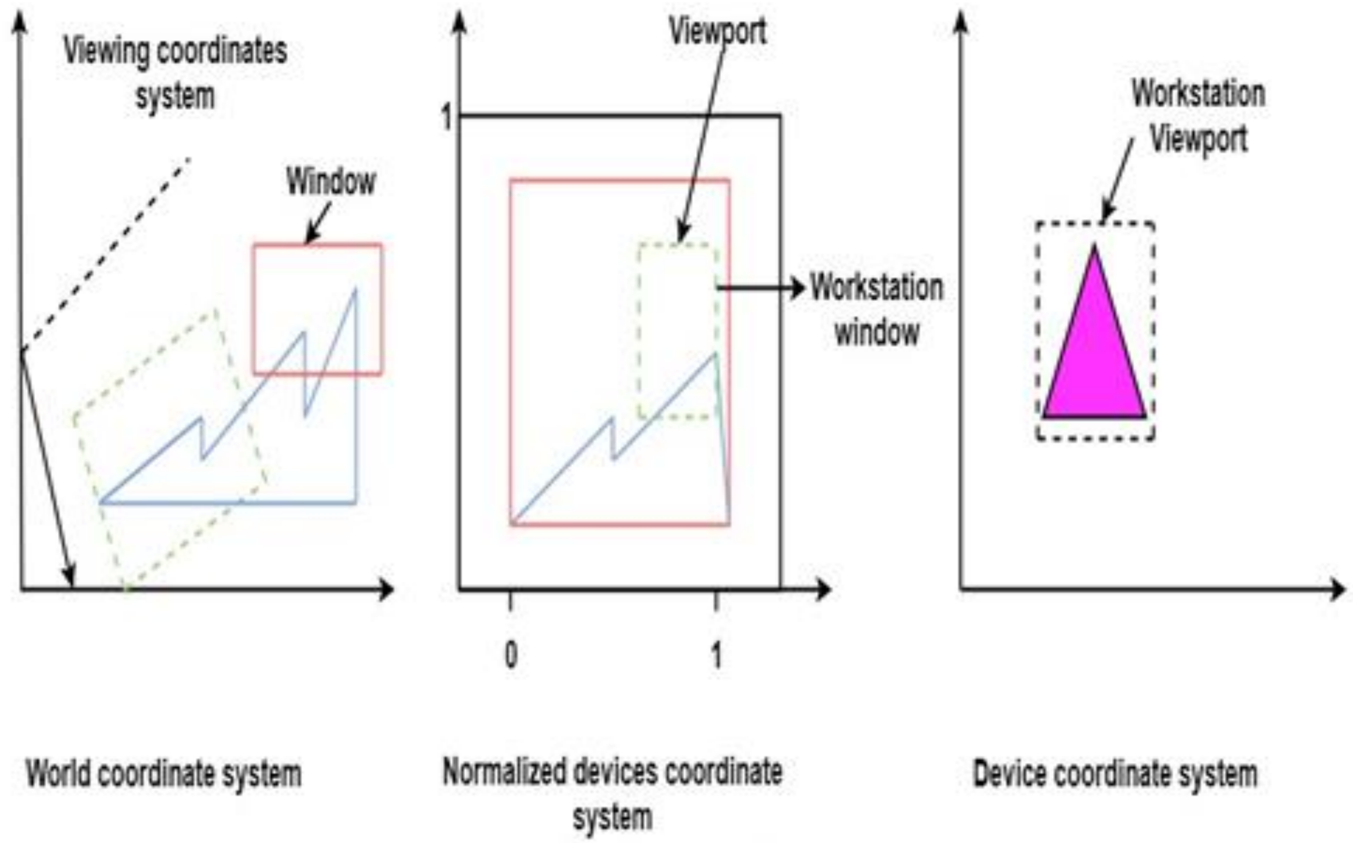


Fig: Zooming effects by mapping different-sized windows on a fixed-size viewport.

3. Normalization and Viewport Transformations

- With some graphics packages, the normalization and window-to-viewport transformations are combined into one operation.
- In this case, the viewport coordinates are often given in the range from 0 to 1 so that the viewport is positioned within a unit square.
- After clipping, the unit square containing the viewport is mapped to the output display device.

Mapping the Clipping Window into a Normalized Viewport

- To illustrate the general procedures for the normalization and viewport transformations, we first consider a viewport defined with normalized coordinate values between 0 and 1.
- Object descriptions are transferred to this normalized space using a transformation that maintains the same relative placement of a point in the viewport as it had in the clipping window.
- If a coordinate position is at the center of the clipping window, for instance, it would be mapped to the center of the viewport.

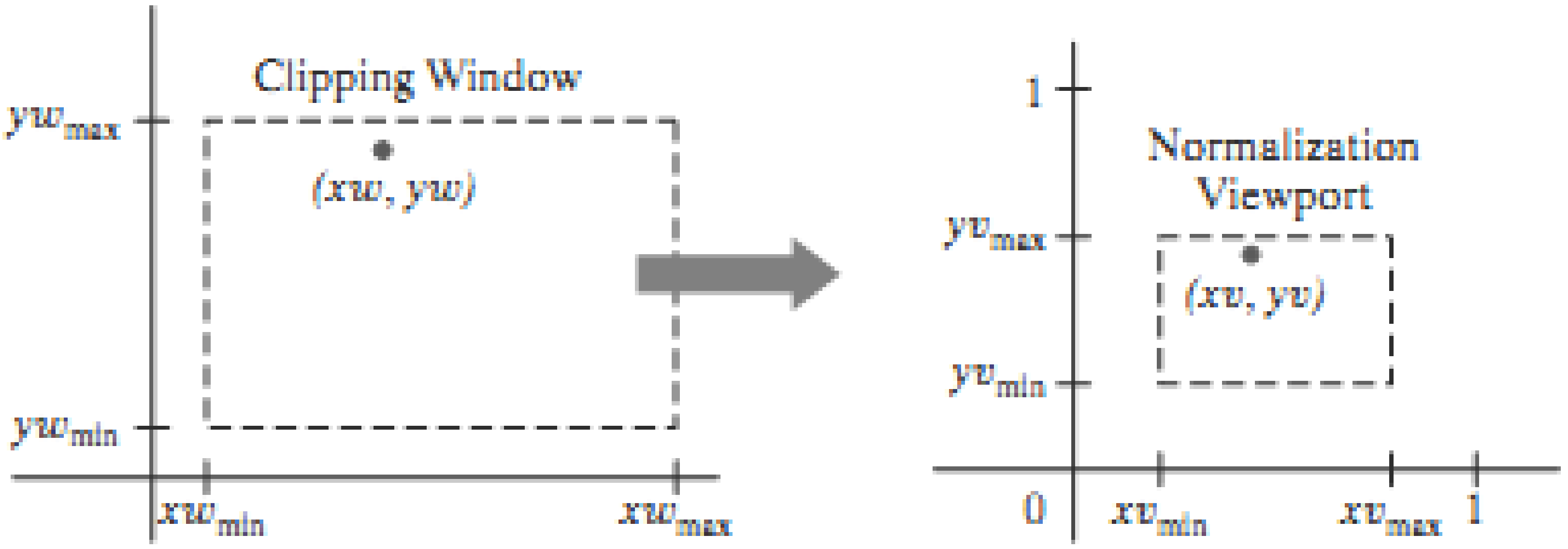


FIGURE 6

A point (xw, yw) in a world-coordinate clipping window is mapped to viewport coordinates (xv, yv), within a unit square, so that the relative positions of the two points in their respective rectangles are the same.

- Position (xw, yw) in the clipping window is mapped to position (xv, yv) in the associated viewport.
- To transform the world-coordinate point into the same relative position within the viewport, we require that:

$$\frac{xv - xv_{\min}}{xv_{\max} - xv_{\min}} = \frac{xw - xw_{\min}}{xw_{\max} - xw_{\min}}$$

$$\frac{yv - yv_{\min}}{yv_{\max} - yv_{\min}} = \frac{yw - yw_{\min}}{yw_{\max} - yw_{\min}}$$

$$xv = s_x xw + t_x$$

$$yv = s_y yw + t_y$$

- and the translation factors are:

$$t_x = \frac{xw_{\max} xv_{\min} - xw_{\min} xv_{\max}}{xw_{\max} - xw_{\min}}$$

$$t_y = \frac{yw_{\max} yv_{\min} - yw_{\min} yv_{\max}}{yw_{\max} - yw_{\min}}$$

- Because we are simply mapping world-coordinate positions into a viewport that is positioned near the world origin, we can also derive using any transformation sequence that converts the rectangle for the clipping window into the viewport rectangle.

Mapping the Clipping Window into a Normalized Square

- Another approach to two-dimensional viewing is to transform the clipping window into a normalized square, clip in normalized coordinates, and then transfer the scene description to a viewport specified in screen coordinates.

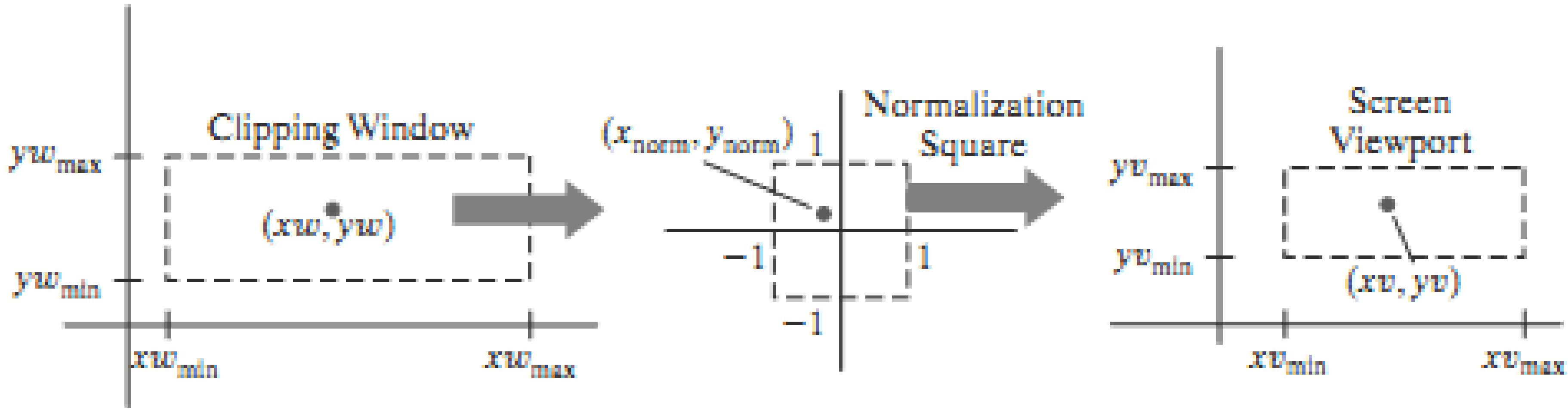


FIGURE 7
A point (xw, yw) in a clipping window is mapped to a normalized coordinate position $(x_{\text{norm}}, y_{\text{norm}})$, then to a screen-coordinate position (xv, yv) in a viewport. Objects are clipped against the normalization square before the transformation to viewport coordinates occurs.

- Making these substitutions in the expressions for tx , ty, sx , and sy, we have:

$$\mathbf{M}_{\text{window, normsquare}} = \begin{bmatrix} \dfrac{2}{xw_{\max} - xw_{\min}} & 0 & -\dfrac{xw_{\max} + xw_{\min}}{xw_{\max} - xw_{\min}} \\ 0 & \dfrac{2}{yw_{\max} - yw_{\min}} & -\dfrac{yw_{\max} + yw_{\min}}{yw_{\max} - yw_{\min}} \\ 0 & 0 & 1 \end{bmatrix}$$

- Similarly, after the clipping algorithms have been applied, the normalized square with edge length equal to 2 is transformed into a specified viewport.

$$\mathbf{M}_{\text{normsquare, viewport}} = \begin{bmatrix} \frac{xv_{\max} - xv_{\min}}{2} & 0 & \frac{xv_{\max} + xv_{\min}}{2} \\ 0 & \frac{yv_{\max} - yv_{\min}}{2} & \frac{yv_{\max} + yv_{\min}}{2} \\ 0 & 0 & 1 \end{bmatrix}$$

- The last step in the viewing process is to position the viewport area in the display window.
- Typically, the lower-left corner of the viewport is placed at a coordinate position specified relative to the lower-left corner of the display window.

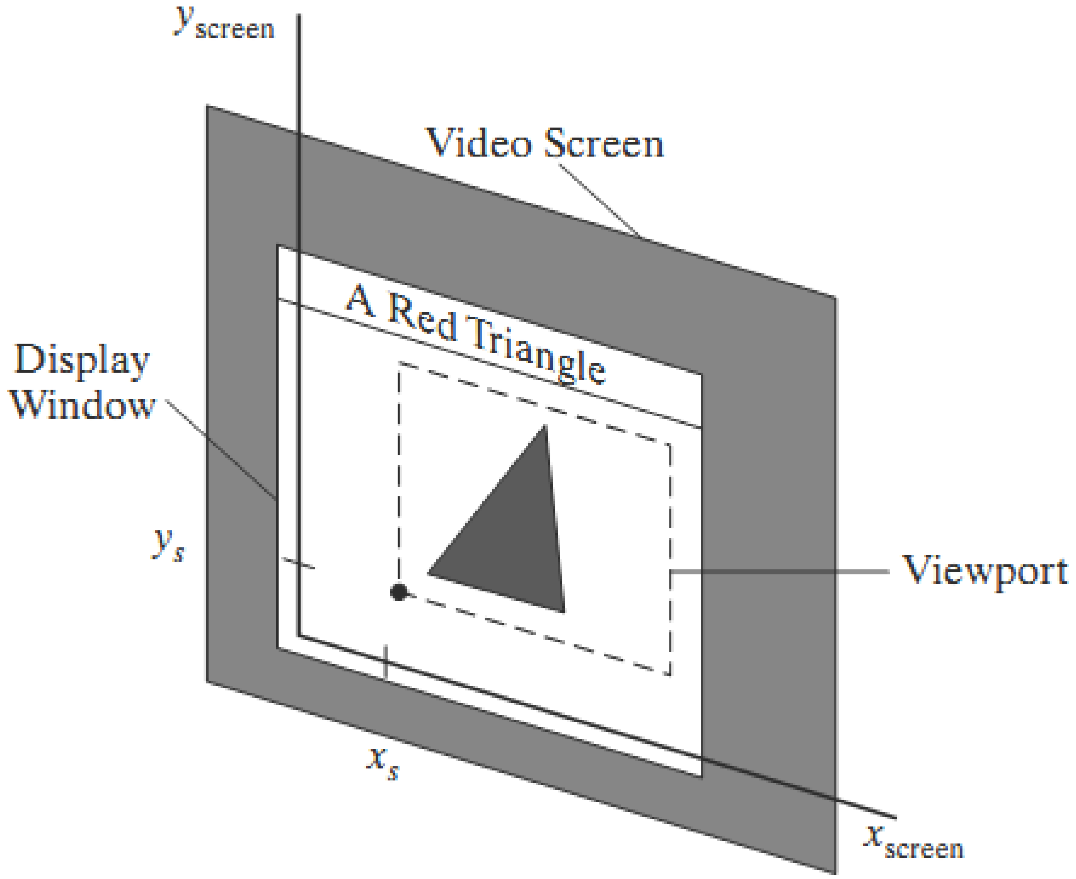


FIGURE 8
A viewport at coordinate position (x_s, y_s) within a display window.

Problems

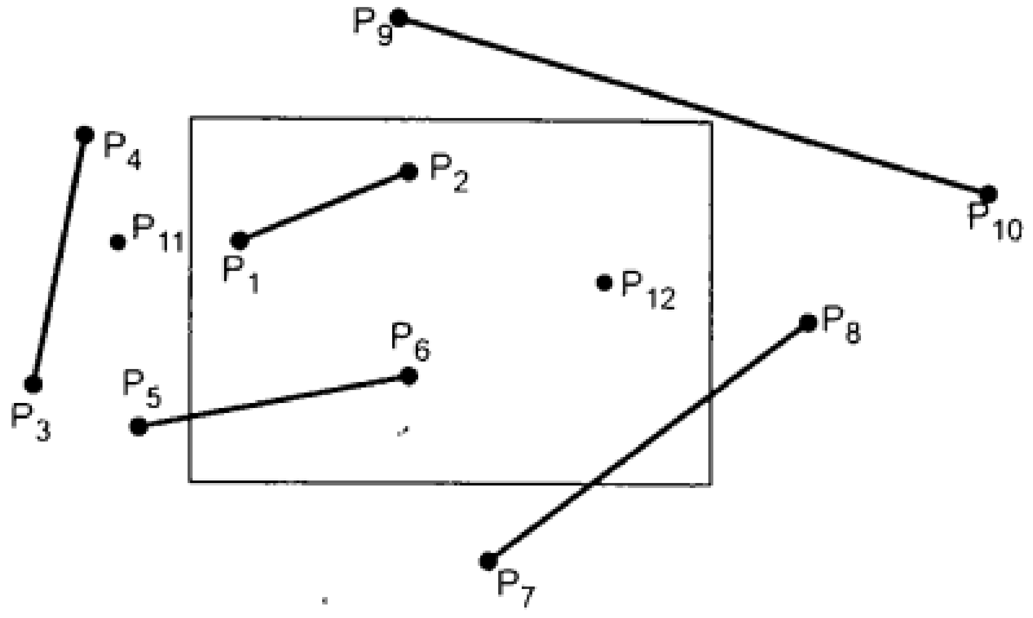
1. Show that 2D reflection through X axis followed by 2D reflection through the line Y = -X is equivalent to a pure rotation about the origin.
2. Prove that successive 2D rotations are additive, i.e. $R(\theta 1) . R(\theta 2) = R(\theta 1 + \theta 2)$
3. Prove that 2D rotation and scaling commute if $Sx = Sy$ or $\theta = n\pi$ for integral n and that otherwise they do not.

4. Clipping Algorithms

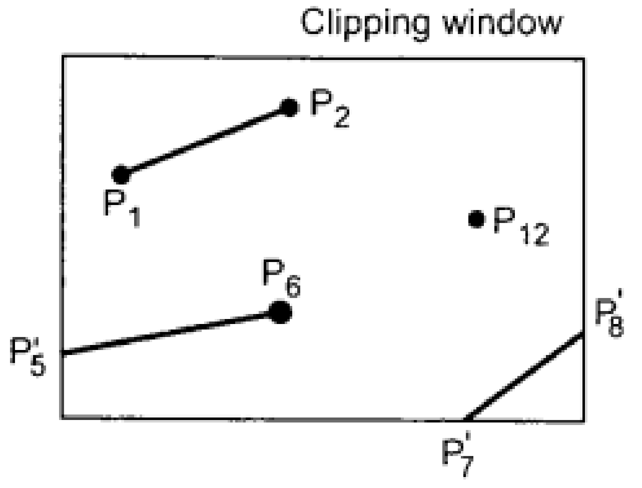
- Generally, any procedure that eliminates those portions of a picture that are either inside or outside a specified region of space is referred to as a clipping algorithm or simply clipping.
- Usually a clipping region is a rectangle in standard position, although we could use any shape for a clipping application.
- The most common application of clipping is in the viewing pipeline, where clipping is applied to extract a designated portion of a scene (either two-dimensional or three-dimensional) for display on an output device.

- Clipping algorithms are applied in two-dimensional viewing procedures to identify those parts of a picture that are within the clipping window.
- Everything outside the clipping window is then eliminated from the scene description that is transferred to the output device for display.
- An efficient implementation of clipping in the viewing pipeline is to apply the algorithms to the normalized boundaries of the clipping window.
- The clipped scene can then be transferred to screen coordinates for final processing.

2D Clipping



(a) Before clipping



(b) After clipping

We explore two-dimensional algorithms for the following:

- Point clipping
- Line clipping (straight-line segments)
- Fill-area clipping (polygons)
- Curve clipping
- Text clipping

I. Point clipping

- For a clipping rectangle in standard position, we save 2D point P = (x, y) for display.
- The points are said to be interior to the clipping window if:

$$x_{w\ min} \le x \le x_{w\ max}$$

$$y_{w\ min} \le y \le y_{w\ max}$$

- If any of these four inequalities is not satisfied, the point is clipped (not saved for display).
- The equal sign indicates that points on the window boundary are included within the window.

II. Line Clipping

- A line-clipping algorithm processes each line in a scene through a series of tests and intersection calculations to determine whether the entire line or any part of it is to be saved.
- The expensive part of a line-clipping procedure is in calculating the intersection positions of a line with the window edges.
- Therefore, a major goal for any line-clipping algorithm is to minimize the intersection calculations.
- To do this, we can first perform tests to determine whether a line segment is completely inside the clipping window or completely outside.

- It is easy to determine whether a line is completely inside a clipping window, but it is more difficult to identify all lines that are entirely outside the window.
- If we are unable to identify a line as completely inside or completely outside a clipping rectangle, we must then perform intersection calculations to determine whether any part of the line crosses the window interior.
- We test a line segment to determine if it is completely inside or outside a selected clipping-window edge by applying the point-clipping tests of the previous section.
- When both endpoints of a line segment are inside all four clipping boundaries, such as the line from P1 to P2 as in figure, the line is completely inside the clipping window and we save it.

2D Line Clipping…

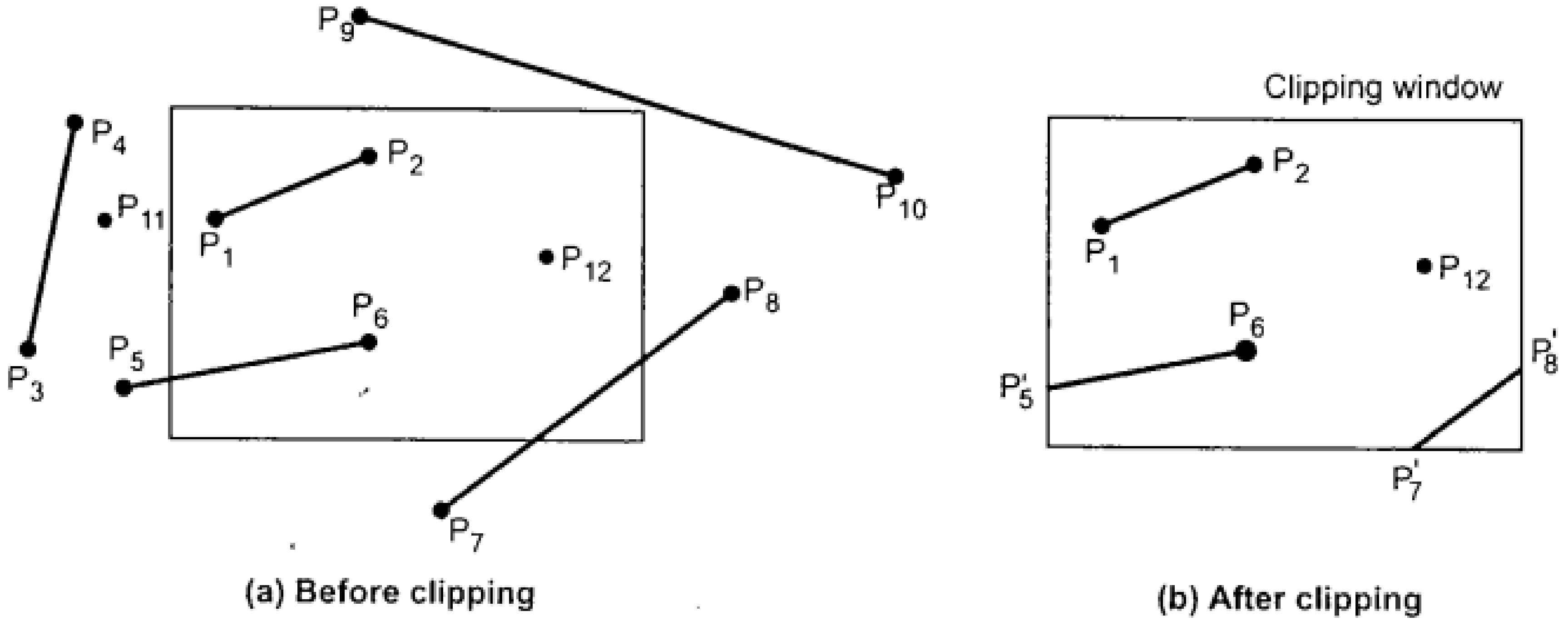


FIGURE 9
Clipping straight-line segments using a standard rectangular clipping window.

- One way to formulate the equation for a straight-line segment is to use the following parametric representation, where the coordinate positions (x0, y0) and (xend, yend) designate the two line endpoints:

$$x = x0 + u(xend - x0)$$

$$y = y0 + u(yend - y0) \qquad 0 \le u \le 1$$

- We can use this parametric representation to determine where a line segment crosses each clipping-window edge by assigning the coordinate value for that edge to either x or y and solving for parameter u.

- And when both endpoints of a line segment are outside any one of the four boundaries (as with line P3 P4 in Figure 9), that line is completely outside the window and it is eliminated from the scene description.

- But if both these tests fail, the line segment intersects at least one clipping boundary and it may or may not cross into the interior of the clipping window.

- However, if the value of u is within the range from 0 to 1, part of the line is inside that border.
- We can then process this inside portion of the line segment against the other clipping boundaries until either we have clipped the entire line or we find a section that is inside the window.
- Processing line segments in a scene using the simple clipping approach described in the preceding paragraph is straightforward, but not very efficient.

II. Line Clipping: Cohen-Sutherland

- This is one of the earliest algorithms to be developed for fast line clipping, and variations of this method are widely used.
- Processing time is reduced in the Cohen-Sutherland method by performing more tests before proceeding to the intersection calculations.
- Initially, every line endpoint in a picture is assigned a four-digit binary value, called a **region code**, and each bit position is used to indicate whether the point is inside or outside one of the clipping-window boundaries.

- For this ordering, the rightmost position (bit 1) references the left clipping-window boundary, and the leftmost position (bit 4) references the top window boundary.
- A value of 1 (or true) in any bit position indicates that the endpoint is outside that window border. Similarly, a value of 0 (or false) in any bit position indicates that the endpoint is not outside (it is inside or on) the corresponding window edge.

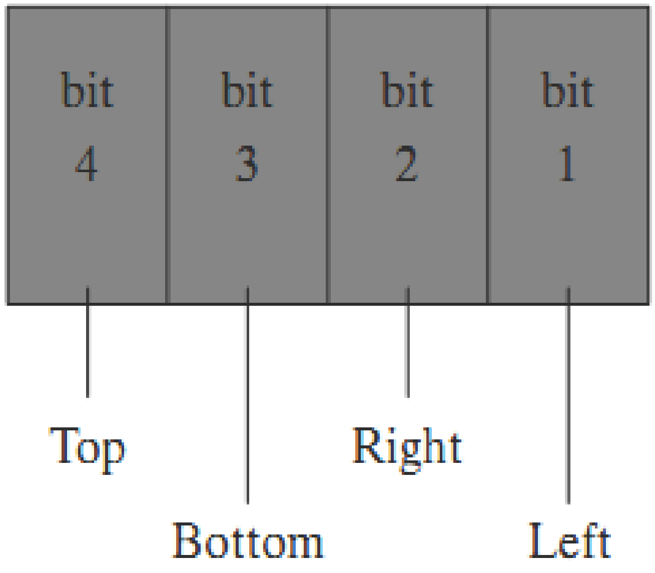
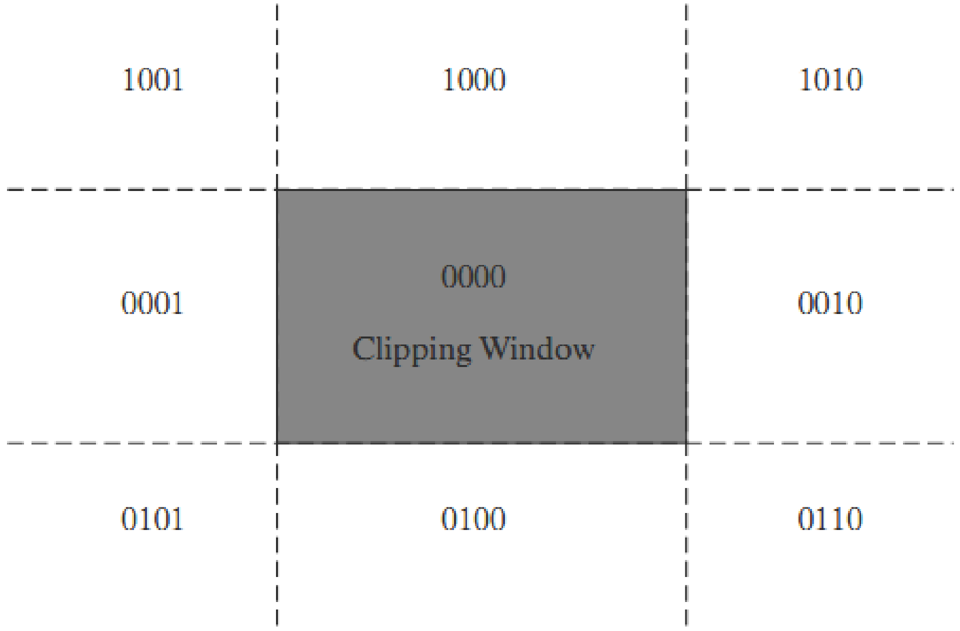


FIGURE 10
A possible ordering for the clipping-window boundaries corresponding to the bit positions in the Cohen-Sutherland endpoint region code.

- Sometimes, a region code is referred to as an “out” code because a value of 1 in any bit position indicates that the spatial point is outside the corresponding clipping boundary.
- Each clipping-window edge divides two-dimensional space into an inside half space and an outside half space.
- Thus, an endpoint that is below and to the left of the clipping window is assigned the region code 0101, and the region-code value for any endpoint inside the clipping window is 0000.



Set Bit 1 – if the end point is to the **left** of the window

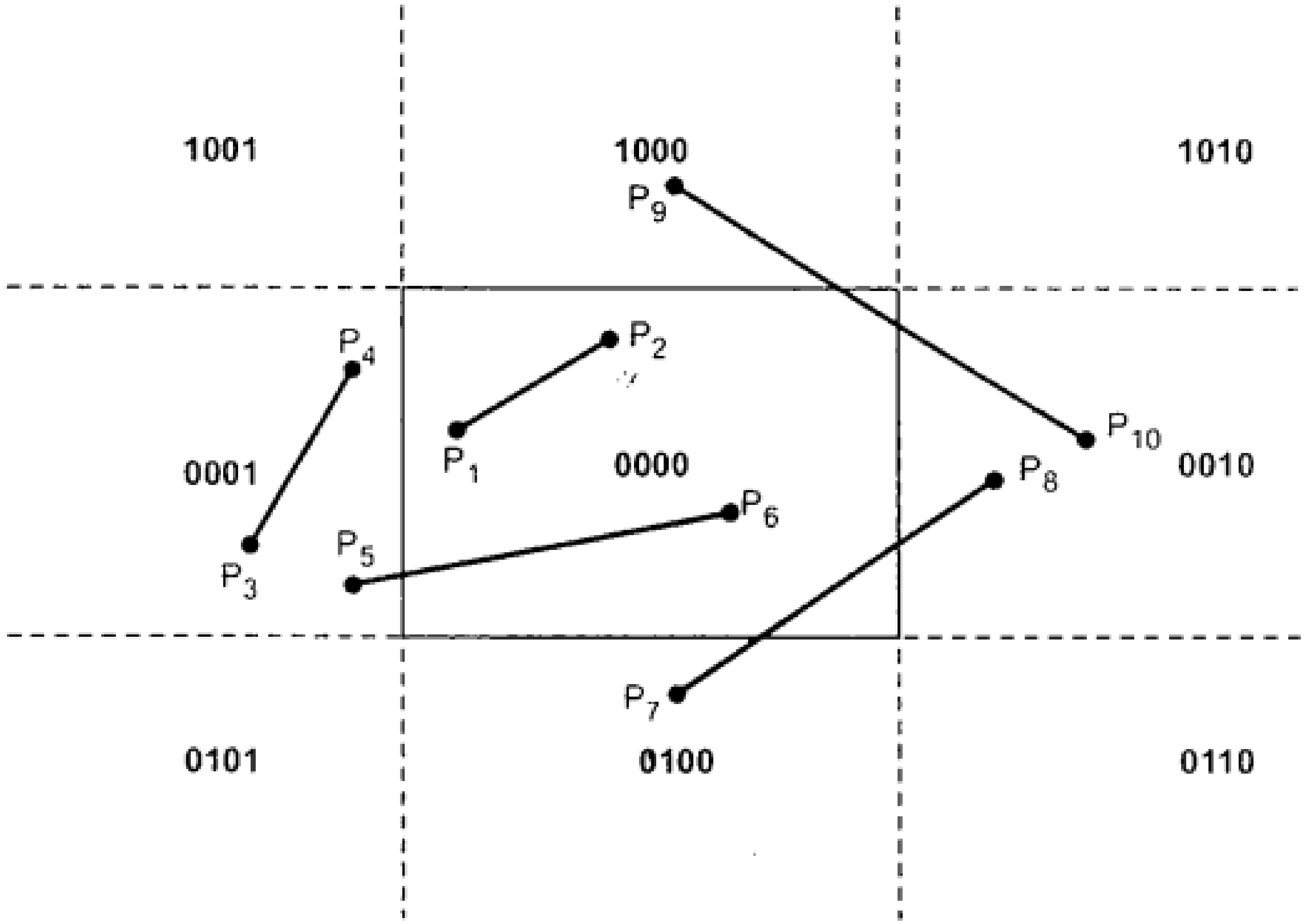
Set Bit 2 – if the end point is to the **right** of the window

Set Bit 3 – if the end point is **below** the window

Set Bit 4 – if the end point is **above** the window

FIGURE 11
The nine binary region codes for identifying the position of a line endpoint, relative to the clipping-window boundaries.

- Bit values in a region code are determined by comparing the coordinate values (x, y) of an endpoint to the clipping boundaries.
- Bit 1 is set to 1 if x < xwmin, and the other three bit values are determined similarly.
- Instead of using inequality testing, we can determine the values for a region-code more efficiently using bit-processing operations and the following two steps:
  - (1) Calculate differences between endpoint coordinates and clipping boundaries.
  - (2) Use the resultant sign bit of each difference calculation to set the corresponding value in the region code.


1001
1000
1010
P_9
P_4
P_2
P_{10}
0001
P_1
0000
P_8
0010
P_6
P_5
P_3
P_7
0101
0100
0110

- Once we have established region codes for all line endpoints, we can quickly determine which lines are completely inside the clip window and which are completely outside.
- Any lines that are completely contained within the window edges have a region code of 0000 for both endpoints, and we save these line segments.
- Any line that has a region-code value of 1 in the same bit position for each endpoint is completely outside the clipping rectangle, and we eliminate that line segment.

- Lines that cannot be identified as being completely inside or completely out-side a clipping window by the region-code tests are next checked for intersection with the window border lines.
- Therefore, several intersection calculations might be necessary to clip a line segment, depending on the order in which we process the clipping boundaries.

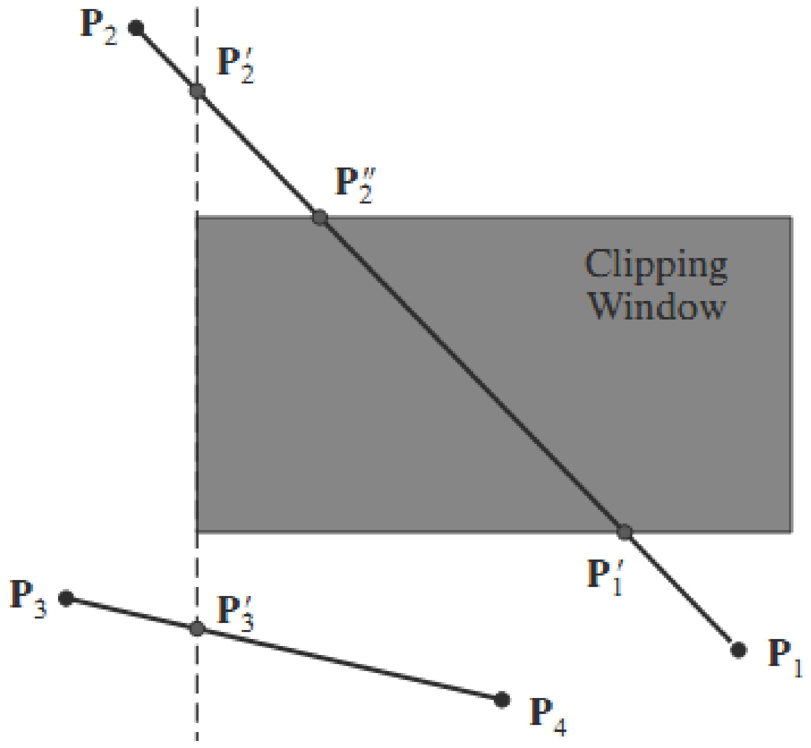


FIGURE 12
Lines extending from one clipping-window region to another may cross into the clipping window, or they could intersect one or more clipping boundaries without entering the window.

- It is possible, when clipping a line segment using this approach, to calculate an intersection position at all four clipping boundaries, depending on how the line endpoints are processed and what ordering we use for the boundaries.
- To determine a boundary intersection for a line segment, we can use the slope-intercept form of the line equation.
- For a line with endpoint coordinates (x0, y0)and (xend, yend), the y coordinate of the intersection point with a vertical clipping border line can be obtained with the calculation: $y = y_0 + m(x - x_0)$

- Similarly, if we are looking for the intersection with a horizontal border, the x coordinate can be calculated as:

$$x = x_0 + \frac{y - y_0}{m}$$

- with y set either to ywmin or to ywmax.

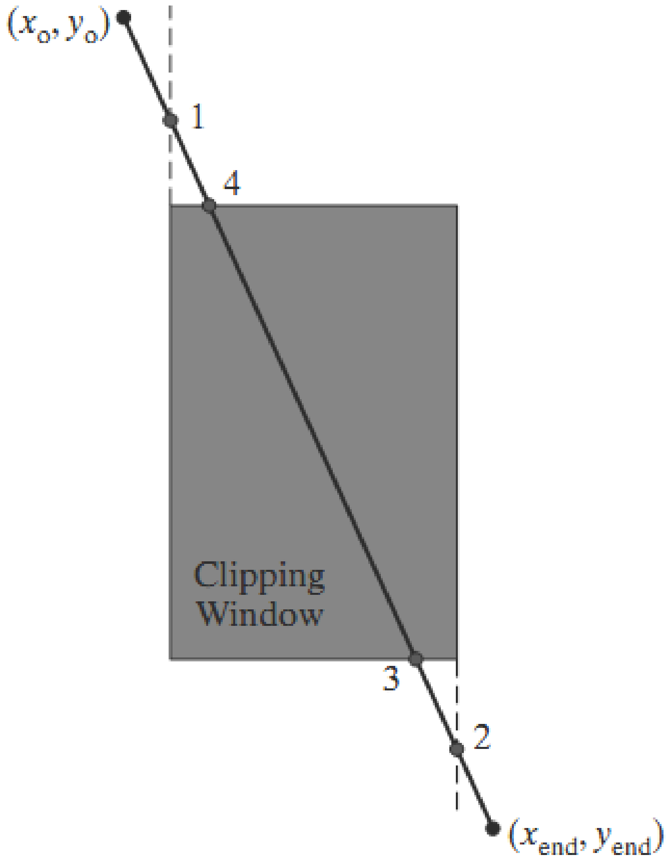


FIGURE 13
Four intersection positions (labeled from 1 to 4) for a line segment that is clipped against the window boundaries in the order left, right, bottom, top.

Cohen-Sutherland Line Clipping Algorithm

1. Read two end points of the line say P_1 (x_1, y_1) and P_2 (x_2, y_2).
2. Read two corners (left-top and right-bottom) of the window, say (Wx_1, Wy_1 and Wx_2, Wy_2).
3. Assign the region codes for two endpoints P_1 and P_2 using following steps :

   Initialize code with bits 0000

   Set Bit 1 – if $(x < Wx_1)$

   Set Bit 2 – if $(x > Wx_2)$

   Set Bit 3 – if $(y < Wy_2)$

   Set Bit 4 – if $(y > Wy_1)$
4. Check for visibility of line P_1 P_2

   a) If region codes for both endpoints P_1 and P_2 are zero then the line is completely visible. Hence draw the line and go to step 9.

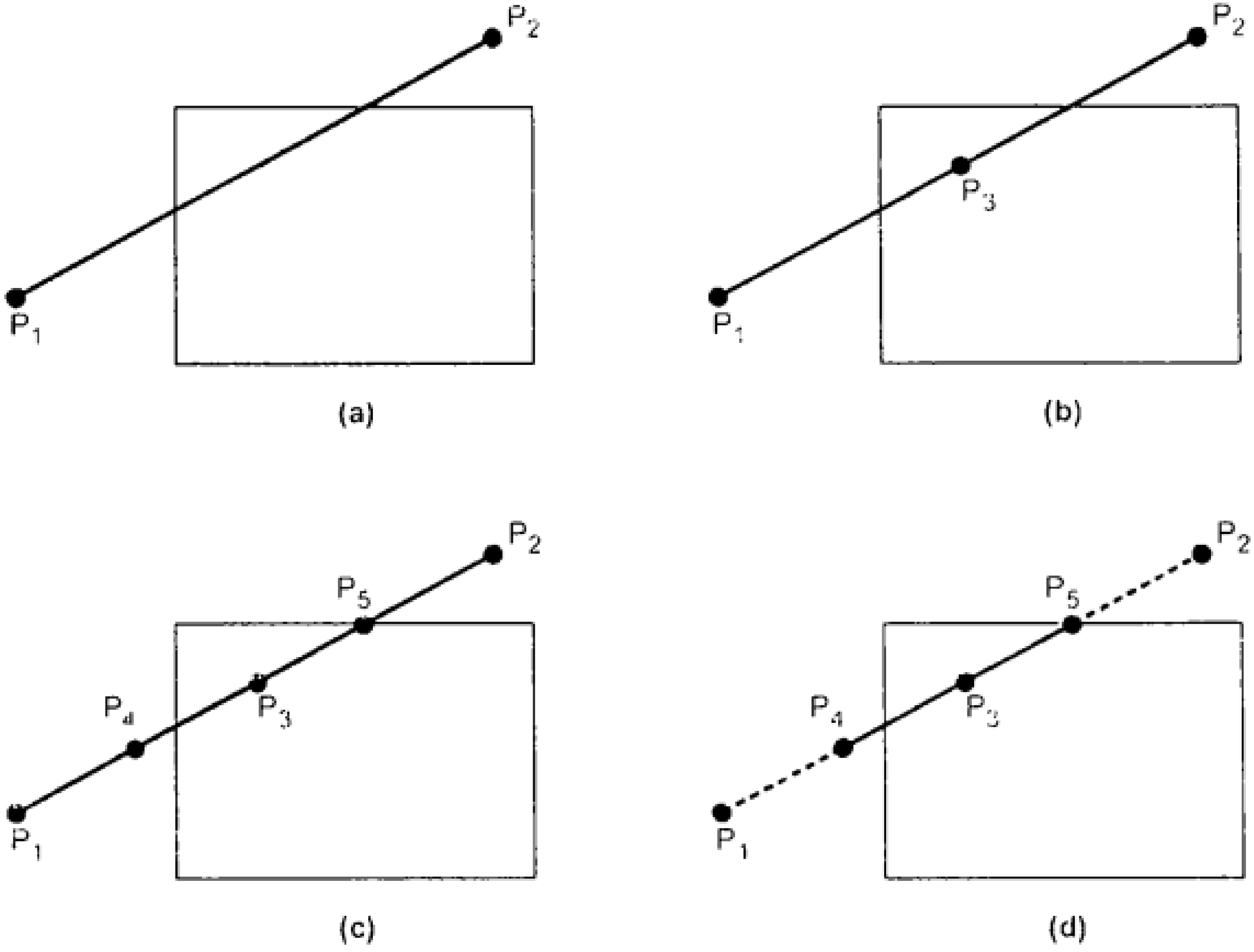
   b) If region codes for endpoints are not zero and the logical ANDing of them is also nonzero then the line is completely invisible, so reject the line and go to step 9.

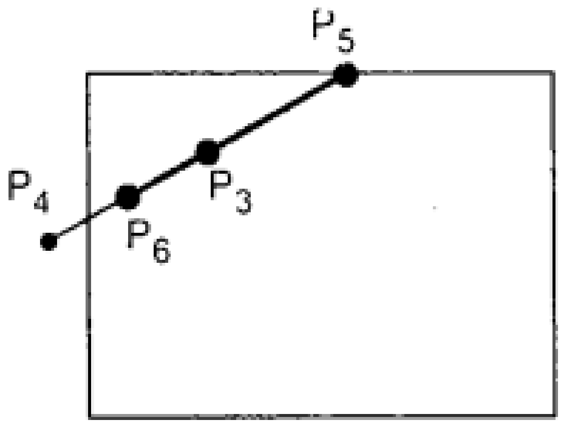
   c) If region codes for two endpoints do not satisfy the conditions in 4a) and 4b)the line is partially visible.

5. Determine the intersecting edge of the clipping window by inspecting the region codes of two endpoints.
   a) If region codes for both the end points are non-zero, find intersection points P_1' and P_2' with boundary edges of clipping window with respect to point P_1 and point P_2, respectively
   b) If region code for any one end point is non zero then find intersection point P_1' or P_2' with the boundary edge of the clipping window with respect to it.
6. Divide the line segments considering intersection points.
7. Reject the line segment if any one end point of it appears outsides the clipping window.
8. Draw the remaining line segments.
9. Stop.

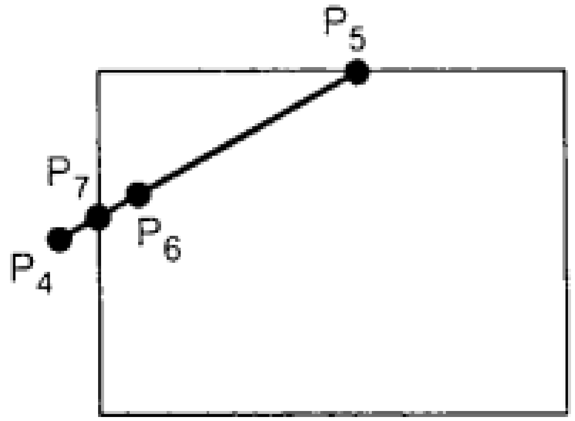
II. Line Clipping: Midpoint Subdivision Algorithm

- The Cohen-Sutherland line clipping algorithm requires the calculation of the intersection of the line with the window edge.
- These calculations can be avoided by repeatedly subdividing the line at its midpoint.
- If line is completely visible it it's drawn and if it is completely invisible it is rejected.
- If the line is partially ble then it is subdivided in two equal parts. The visibility tests are then applied to each half and this subdivision process is repeated until we get completely visible and invisible line segments.

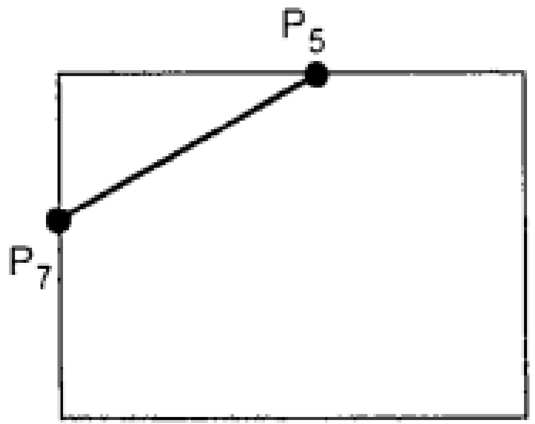

P_1
P_2
P_3
P_4
P_5
(a)
(b)
(c)
(d)



(e)

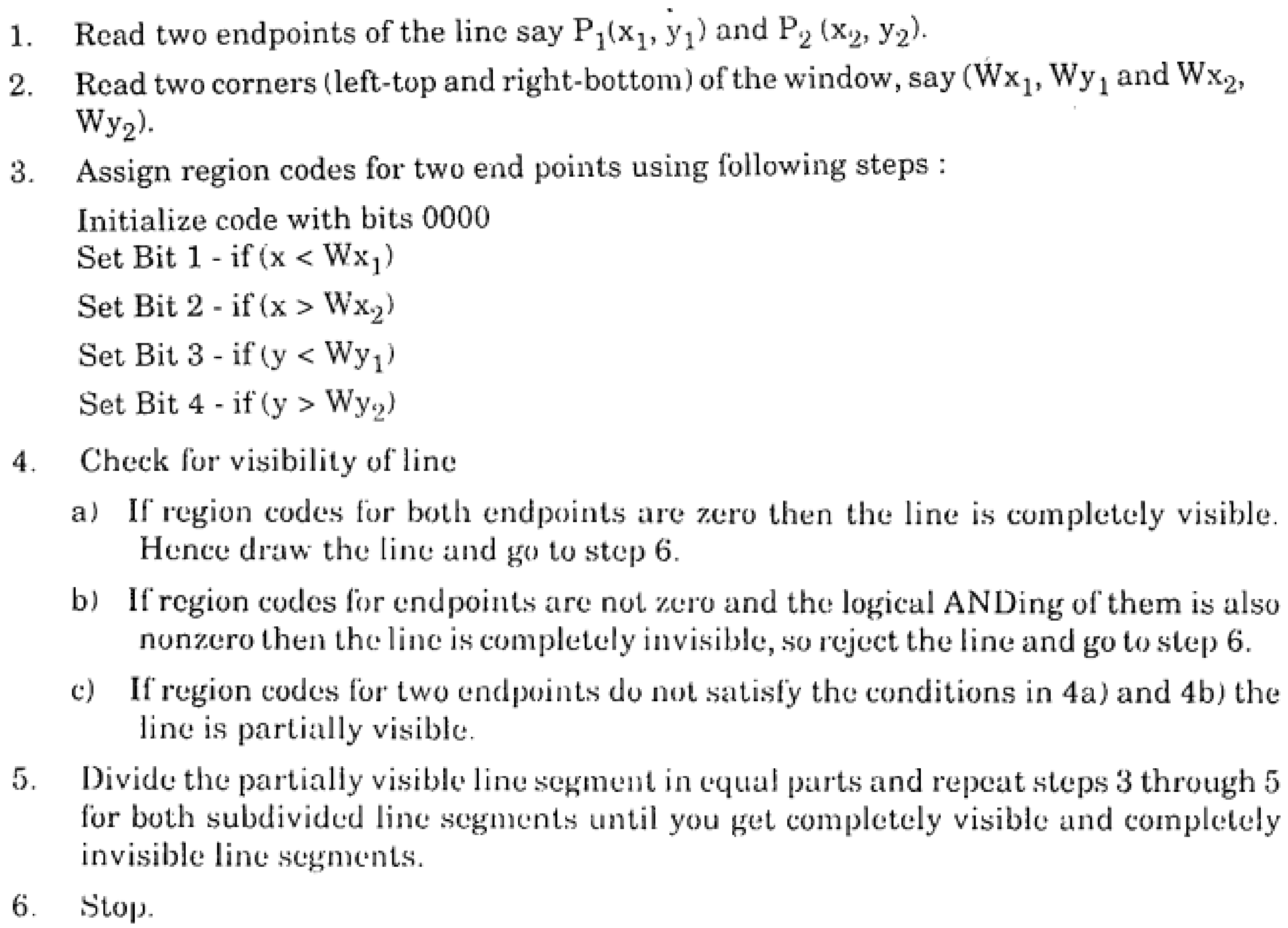


(f)



(g)

Midpoint Subdivision Algorithm:



1. Read two endpoints of the line say $P_1(x_1, y_1)$ and $P_2(x_2, y_2)$.
2. Read two corners (left-top and right-bottom) of the window, say (Wx_1, Wy_1 and Wx_2, Wy_2).
3. Assign region codes for two end points using following steps :

   Initialize code with bits 0000
   Set Bit 1 - if $(x < Wx_1)$
   Set Bit 2 - if $(x > Wx_2)$
   Set Bit 3 - if $(y < Wy_1)$
   Set Bit 4 - if $(y > Wy_2)$
4. Check for visibility of line
   a) If region codes for both endpoints are zero then the line is completely visible. Hence draw the line and go to step 6.
   b) If region codes for endpoints are not zero and the logical ANDing of them is also nonzero then the line is completely invisible, so reject the line and go to step 6.
   c) If region codes for two endpoints do not satisfy the conditions in 4a) and 4b) the line is partially visible.
5. Divide the partially visible line segment in equal parts and repeat steps 3 through 5 for both subdivided line segments until you get completely visible and completely invisible line segments.
6. Stop.

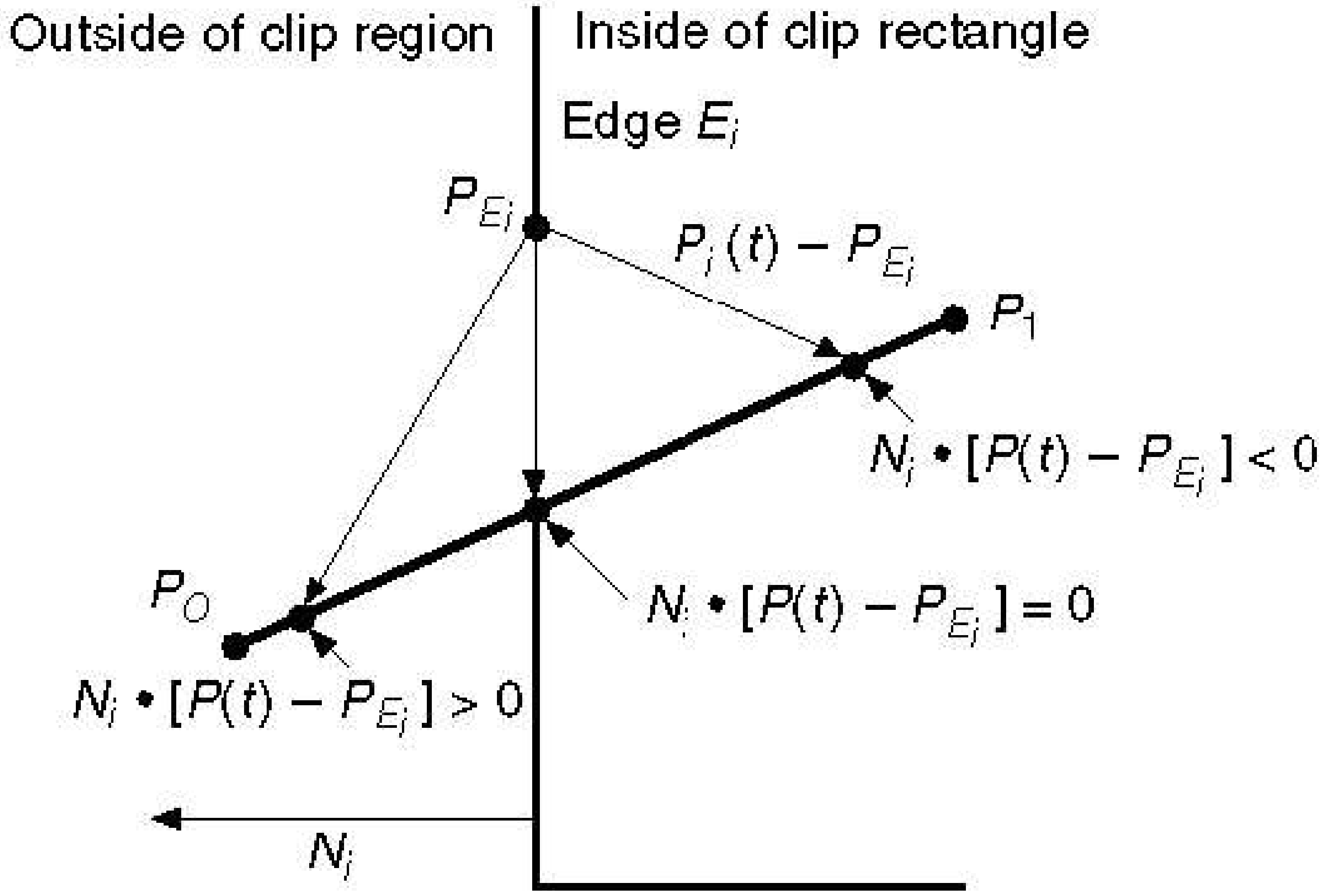
II. Line Clipping: Cyrus-Beck Line Clipping

- The **Cyrus–Beck algorithm** is a generalized line clipping algorithm.

- It was designed to be more efficient than the Cohen-Sutherland algorithm, which uses repetitive clipping.

- Cyrus Beck is a line clipping algorithm that is made for convex polygons.

- It allows line clipping for non-rectangular windows as in Cohen-Sutherland line clipping method.


Outside of clip region
Inside of clip rectangle
Edge E_i
P_{E_i}
$P_i(t) - P_{E_i}$
P_1
$N_i \bullet [P(t) - P_{E_i}] < 0$
$N_i \bullet [P(t) - P_{E_i}] = 0$
P_O
$N_i \bullet [P(t) - P_{E_i}] > 0$
N_i

- Parametric line equation P_0P_1:$P(t) = P_0 + t(P_1 - P_0)$
- Let N_i be the outward normal edge E_i. Now pick any arbitrary point P_{Ei} on edge E_i then the dot product N_i.[P*[Math Processing Error]t* – P_{Ei}] determines whether the point P*[Math Processing Error]t* is "inside the clip edge" or "outside" the clip edge or "on" the clip edge.
  - The point P*[Math Processing Error]t* is **inside** if N_i.[P*[Math Processing Error]t* – P_{Ei}] < 0
  - The point P*[Math Processing Error]t* is **outside** if N_i.[P*[Math Processing Error]t* – P_{Ei}] > 0
  - The point P*[Math Processing Error]t* is **on the edge** if N_i.[P*[Math Processing Error]t* – P_{Ei}] = 0

- N_i.[P*[Math Processing Error]t* – P_{Ei}] = 0
- N_i.[P_0 + t(P_1 - P_0) – P_{Ei}] = 0 *[Math Processing Error]ReplacingP(t* with P_0 + t(P_1 - P_0))
- N_i.[P_0 – P_{Ei}] + N_i.t[P_1 - P_0] = 0
- N_i.[P_0 – P_{Ei}] + $N_i \cdot$tD = 0 (substituting D for [P_1 - P_0])
- N_i.[P_0 – P_{Ei}] = - $N_i \cdot$tD

- It is valid for the following conditions −
  - $N_i \neq 0$ *[Math Processing Error]*
  - $D \neq 0$ ($P_1 \neq P_0$)
  - $N_i \cdot D \neq 0$ (P_0P_1 not parallel to E_i)

Cyrus-Beck Line Clipping Algorithm:

```
1. Read two end points of the line, say $P_1$ and $P_2$
2. Read vertex coordinates of the clipping window
3. Calculate $D = P_2 - P_1$
4. Assign boundary point (f) with particular edge
5. Find inner normal vector for corresponding edge
6. Calculate $D \cdot n$ and $W = P_1 - f$
7. If $D \cdot n > 0$
       $t_L = - \frac{W \cdot n}{D \cdot n}$
   else
       $t_U = - \frac{W \cdot n}{D \cdot n}$
   end if
```

8. Repeat steps 4 through 7 for each edge of the clipping window
9. Find maximum lower limit and minimum upper limit
10. If maximum lower limit and minimum upper limit do not satisfy condition $0 \le t \le 1$ then ignore the line.
11. Calculate the intersection points by substituting values of maximum lower limit and minimum upper limit in the parametric equation of the line P_1P_2.
12. Draw the line segment $P(t_L)$ to $P(t_U)$.
13. Stop.

II. Line Clipping: Liang-Barsky Line Clipping

- Faster line-clipping algorithms have been developed that do more line testing before proceeding to the intersection calculations.
- One of the earliest efforts in this direction is an algorithm developed by Cyrus and Beck, which is based on analysis of the parametric line equations.
- Later, Liang and Barsky independently devised an even faster form of the parametric line-clipping algorithm.

- For a line segment with endpoints (x0, y0) and (xend, yend), we can describe the line with the parametric form. $x = x_0 + u\Delta x$

$$y = y_0 + u\Delta y \qquad 0 \leq u \leq 1$$

- In the Liang-Barsky algorithm, the parametric line equations are combined with the point-clipping conditions to obtain the inequalities:

$$xw_{\min} \leq x_0 + u\Delta x \leq xw_{\max}$$
$$yw_{\min} \leq y_0 + u\Delta y \leq yw_{\max}$$

which can be expressed as

$$u\, p_k \leq q_k, \qquad k = 1, 2, 3, 4$$

where parameters p and q are defined as

Left	$p_1 = -\Delta x,$	$q_1 = x_0 - xw_{\min}$
Right	$p_2 = \Delta x,$	$q_2 = xw_{\max} - x_0$
Bottom	$p_3 = -\Delta y,$	$q_3 = y_0 - yw_{\min}$
Top	$p_4 = \Delta y,$	$q_4 = yw_{\max} - y_0$

- Any line that is parallel to one of the clipping-window edges has pk = 0 for the value of k corresponding to that boundary, where k = 1, 2, 3, and 4 correspond to the left, right, bottom, and top boundaries, respectively.

- We can define inequalities as:

  $x_1 + u\Delta x >= x_{wmin}$

  $x_1 + u\Delta x <= x_{wmax}$

  $y_1 + u\Delta y >= y_{wmin}$

  $y_1 + u\Delta y <= y_{wmax}$

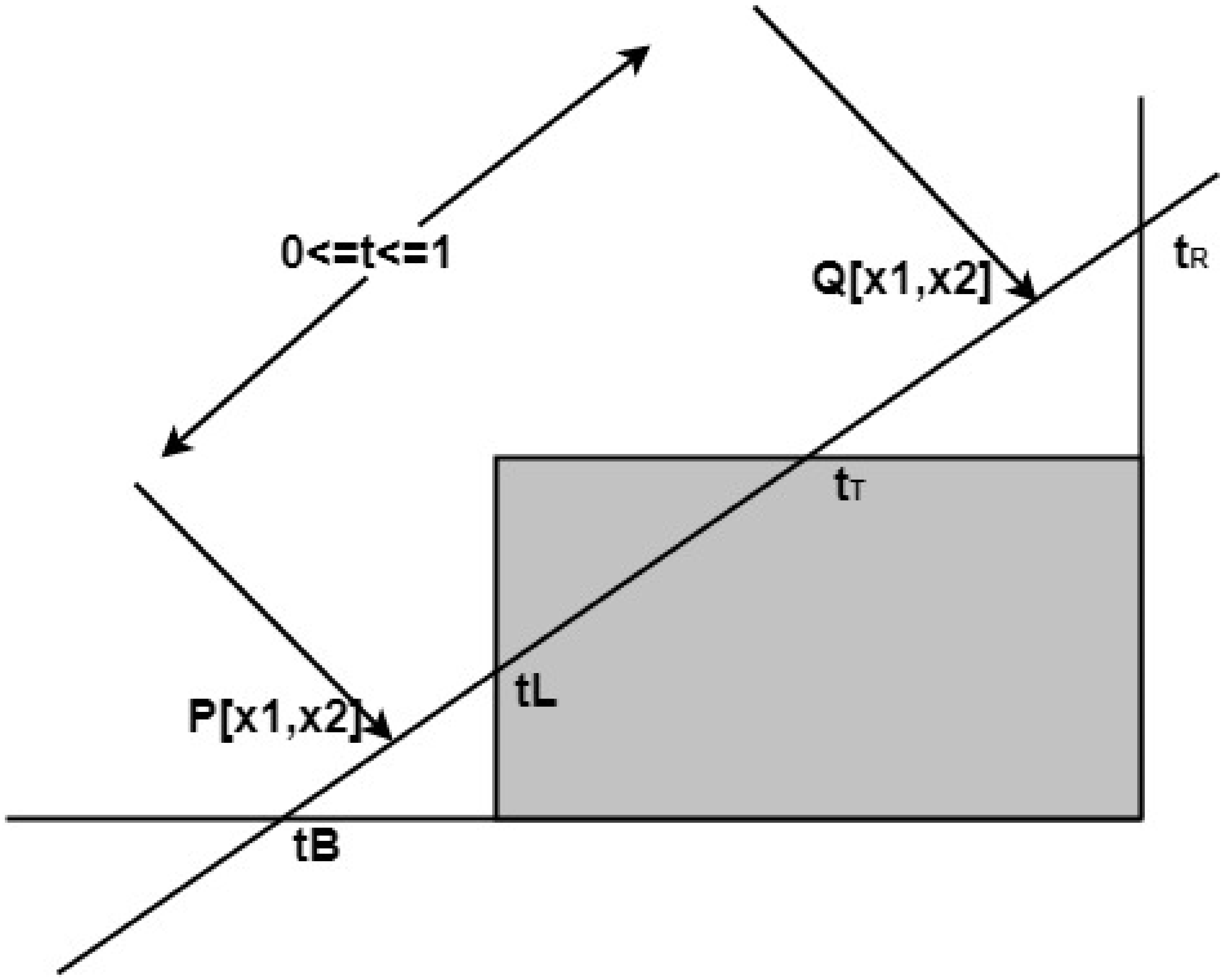
- Now we can expressed as:

  $u\Delta x >= x_{wmin} - x_1$ Change

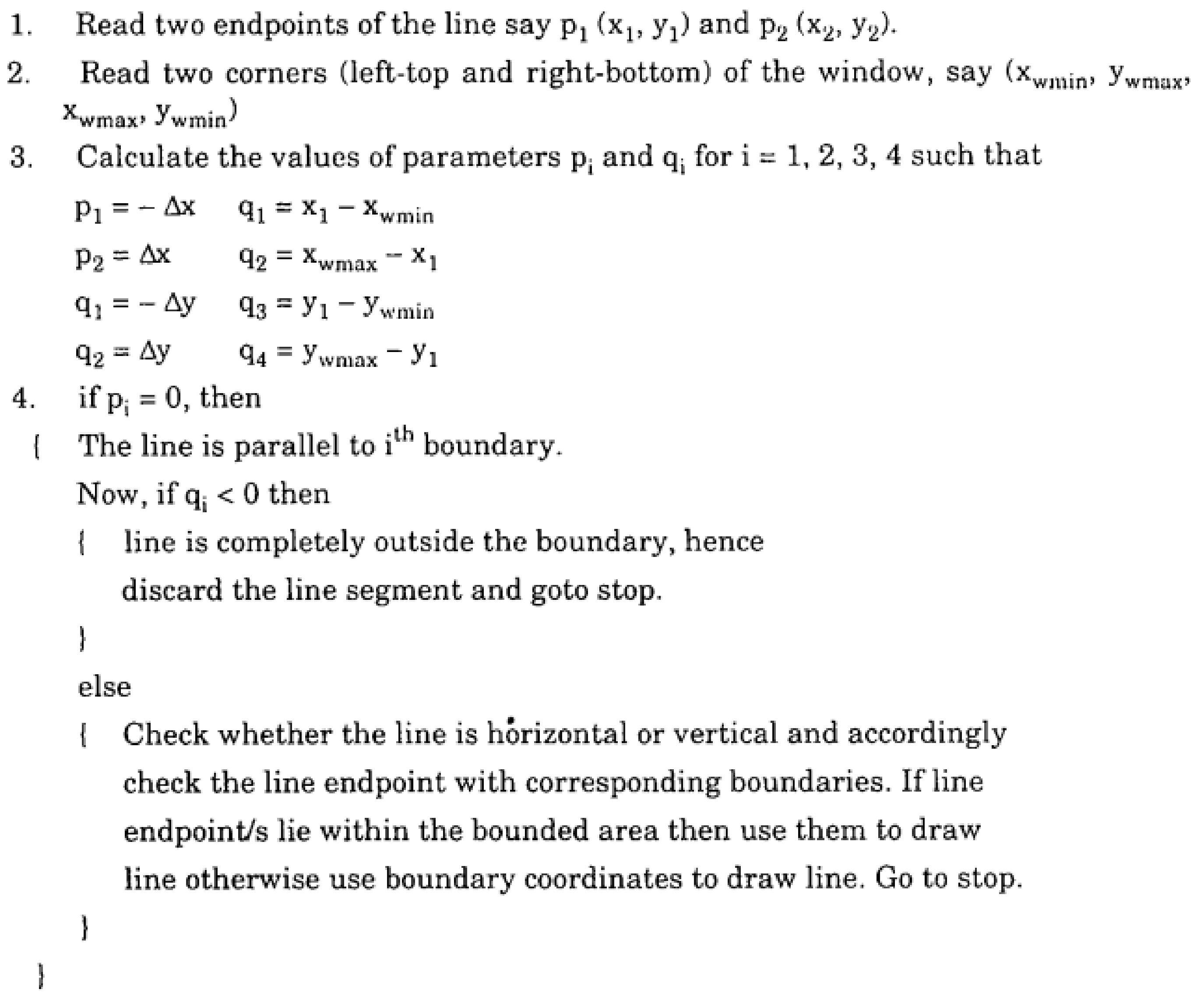
  $u\Delta x <= x_{wmax} - x_1$ Ok

  $u\Delta y >= y_{wmin} - y_1$ Change

  $u\Delta y <= y_{wmax} - y_1$ Ok


0<=t<=1
Q[x1,x2]
t_R
t_T
tL
P[x1,x2]
tB

Liang-Barsky Line Clipping Algorithm



```
1.  Read two endpoints of the line say $p_1 (x_1, y_1)$ and $p_2 (x_2, y_2)$.
2.  Read two corners (left-top and right-bottom) of the window, say ($x_{wmin}$, $y_{wmax}$,
    $x_{wmax}$, $y_{wmin}$)
3.  Calculate the values of parameters $p_i$ and $q_i$ for i = 1, 2, 3, 4 such that
    $p_1 = -\Delta x$    $q_1 = x_1 - x_{wmin}$
    $p_2 = \Delta x$     $q_2 = x_{wmax} - x_1$
    $q_1 = -\Delta y$    $q_3 = y_1 - y_{wmin}$
    $q_2 = \Delta y$     $q_4 = y_{wmax} - y_1$
4.  if $p_i = 0$, then
  { The line is parallel to $i^{th}$ boundary.
    Now, if $q_i < 0$ then
    {   line is completely outside the boundary, hence
        discard the line segment and goto stop.
    }
    else
    {   Check whether the line is horizontal or vertical and accordingly
        check the line endpoint with corresponding boundaries. If line
        endpoint/s lie within the bounded area then use them to draw
        line otherwise use boundary coordinates to draw line. Go to stop.
    }
  }
```

```
5.   Initialise values for $t_1$ and $t_2$ as
     $t_1 = 0$ and $t_2 = 1$
6.   Calculate values for $q_i/p_i$ for $i = 1, 2, 3, 4$
7.   Select values of $q_i/p_i$ where $p_i < 0$ and assign maximum out of them as $t_1$.
8.   Select values of $q_i/p_i$ where $p_i > 0$ and assign minimum out of them as $t_2$.
9.   If ($t_1 < t_2$)
  {  Calculate the endpoints of the clipped line as follows :
     $xx_1 = x_1 + t_1 \Delta x$
```

Advantages:

- It is more efficient that Cohen-Sutherland algorithm, since intersection calculations are reduced.
- It requires only one division to update parameters t1 and t2.
- Window intersections of the line are computed only once.

II. Line Clipping: Nicholl-Lee-Nicholl Line Clipping

- Theory Assignment

III. Polygon Fill-area clipping

- Graphics packages typically support only fill areas that are polygons, and often only convex polygons.
- To clip a polygon fill area, we cannot apply a line-clipping method to the individual polygon edges directly because this approach would not work.
- What we require is a procedure that will output one or more closed polylines for the boundaries of the clipped fill area, so that polygons can be scan-converted to fill the interiors with the assigned color or pattern.

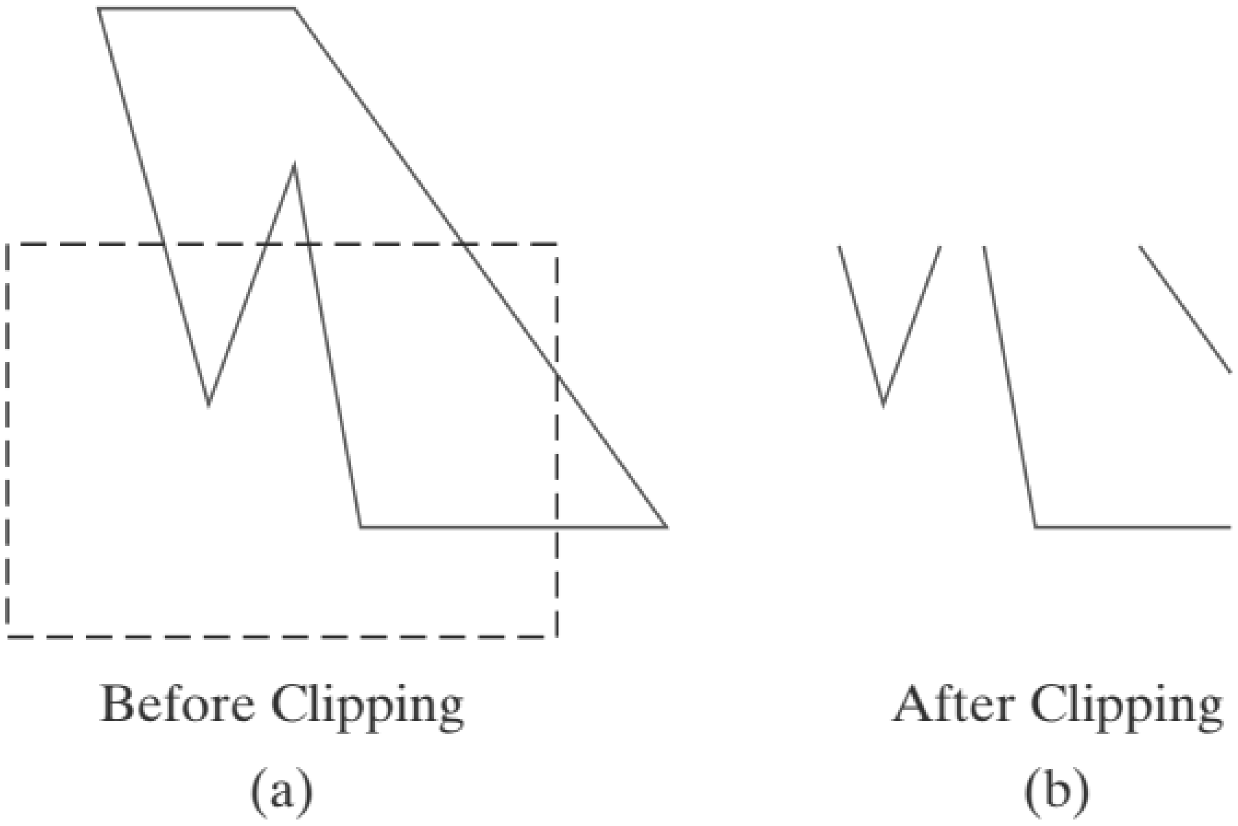


FIGURE 19
A line-clipping algorithm applied to
the line segments of the polygon boundary in (a)
generates the unconnected set of lines in (b).

- We can process a polygon fill area against the borders of a clipping window using the same general approach as in line clipping.

- We need to maintain a fill area as an entity as it is processed through the clipping stages.

- Thus, we can clip a polygon fill area by determining the new shape for the polygon as each clipping-window edge is processed.

- The interior fill for the polygon would not be applied until the final clipped border has been determined.

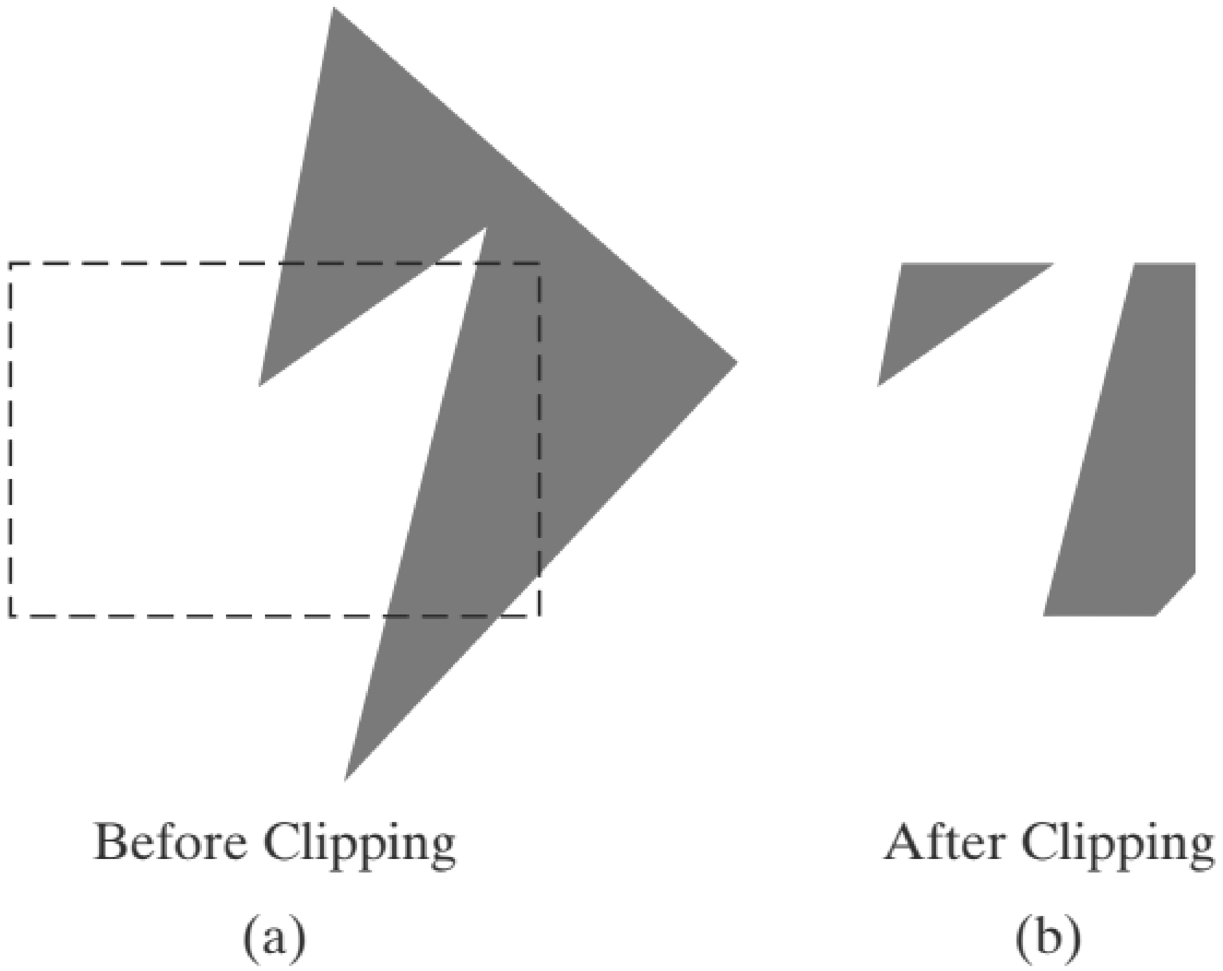


FIGURE 20
Display of a correctly clipped polygon fill area.

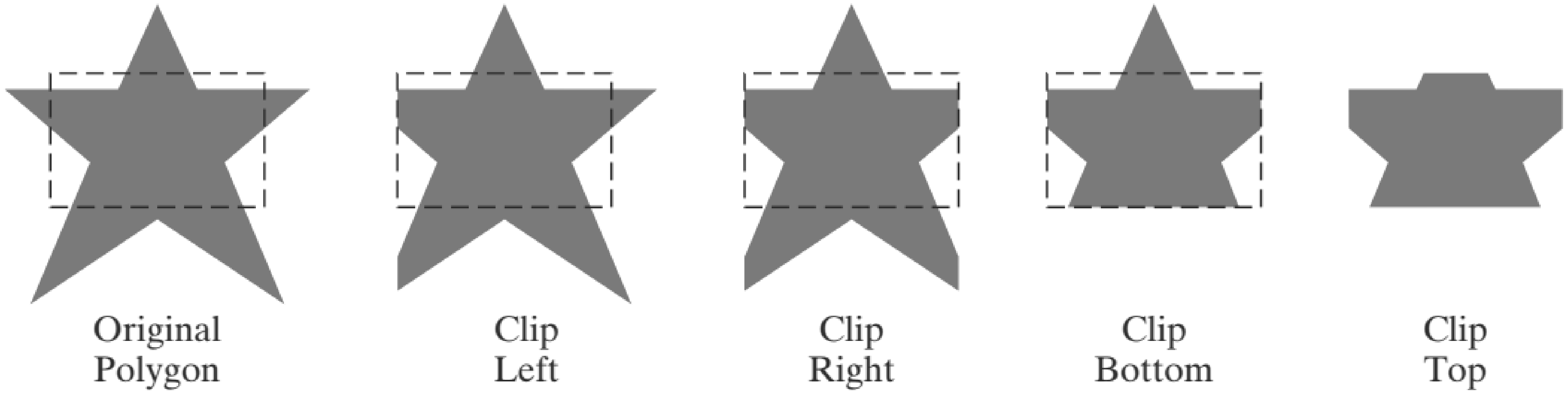


FIGURE 21
Processing a polygon fill area against successive clipping-window boundaries.

- We first tested a line segment to determine whether it could be completely saved or completely clipped, that will be the same with a polygon fill area by checking its coordinate extents.

- If the minimum and maximum coordinate values for the fill area are inside all four clipping boundaries, the fill area is saved for further processing.

- If these coordinate extents are all outside any of the clipping-window borders, we eliminate the polygon from the scene description.

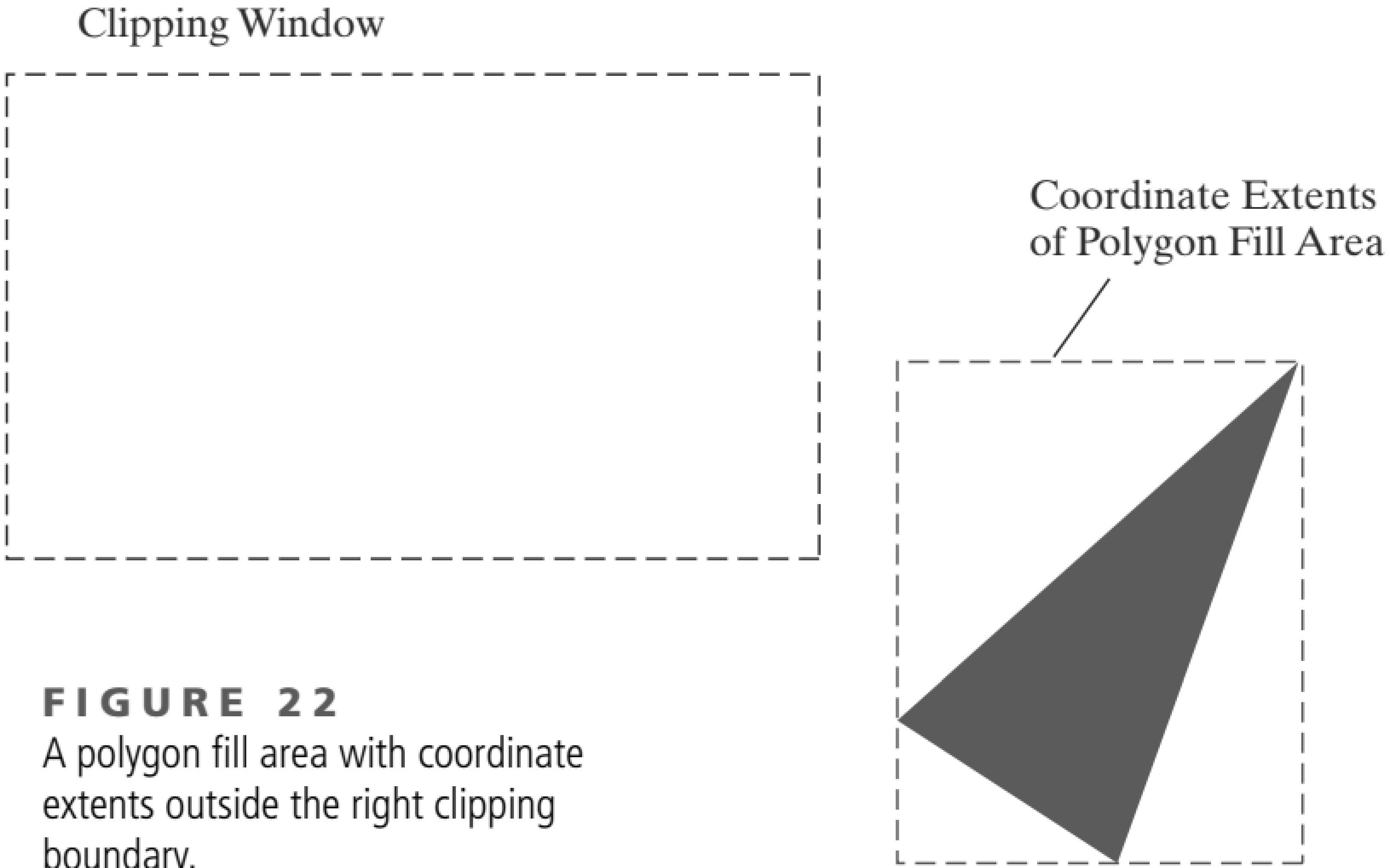


FIGURE 22
A polygon fill area with coordinate extents outside the right clipping boundary.

- When we cannot identify a fill area as being completely inside or completely outside the clipping window, we then need to locate the polygon intersection positions with the clipping boundaries.
- One way to implement convex-polygon clipping is to create a new vertex list at each clipping boundary, and then pass this new vertex list to the next boundary clipper.
- The output of the final clipping stage is the vertex list for the clipped polygon.
- For concave-polygon clipping, we would need to modify this basic approach so that multiple vertex lists could be generated.

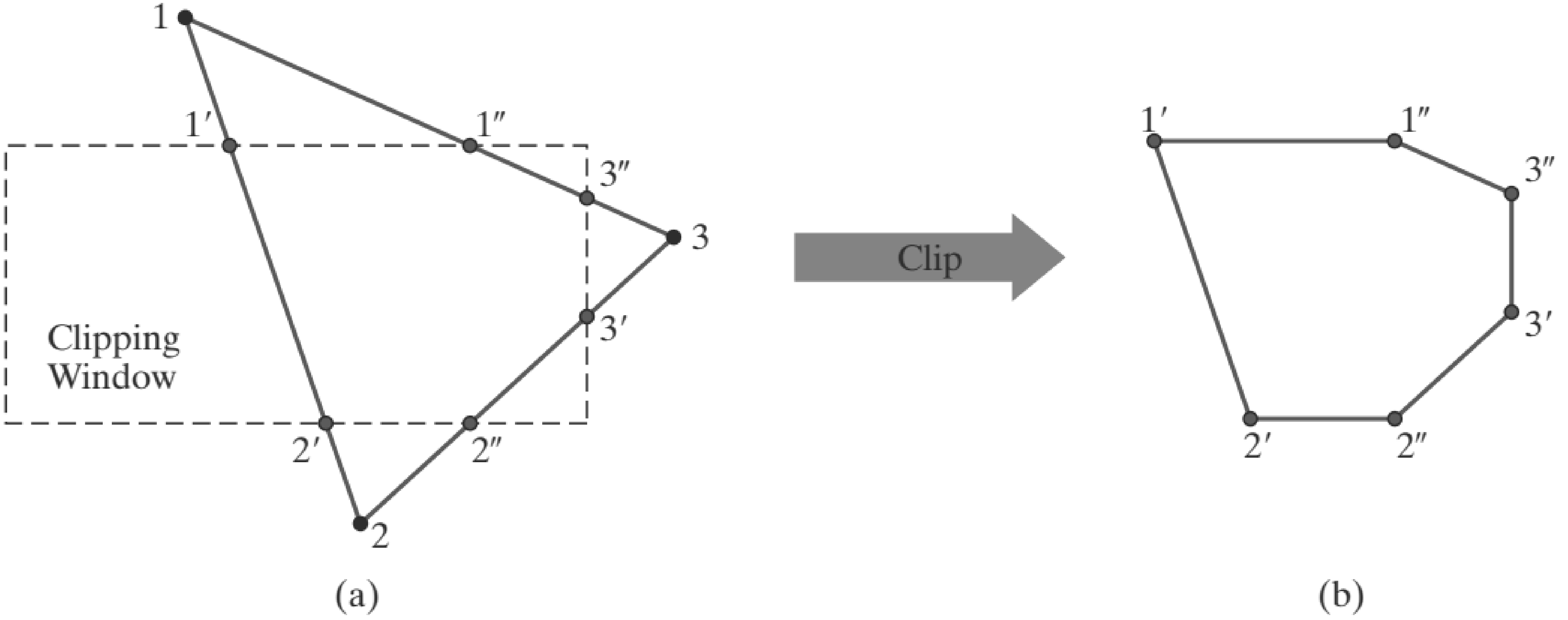


FIGURE 23
A convex-polygon fill area (a), defined with the vertex list {1, 2, 3}, is clipped to produce the fill-area shape shown in (b), which is defined with the output vertex list {1′, 2′, 2″, 3′, 3″, 1″}.

1. Polygon Clipping: **Sutherland-Hodgeman**

- To send the polygon vertices through each clipping stage so that a single clipped vertex can be immediately passed to the next stage.
- This eliminates the need for an output set of vertices at each clipping stage, and it allows the boundary-clipping routines to be implemented in parallel.
- The final output is a list of vertices that describe the edges of the clipped polygon fill area.

- The general strategy is to send the pair of endpoints for each successive polygon line segment through the series of clippers (left, right, bottom, and top).
- As soon as clipper completes the processing of one pair of vertices, the clipped coordinate values, if any, for that edge are sent to the next clipper.
- Then the first clipper processes the next pair of endpoints.
- There are four possible cases that need to be considered when processing a polygon edge against one of the clipping boundaries.

I. One possibility is that the first edge endpoint is outside the clipping boundary and the second endpoint is inside.
II. Or, both endpoints could be inside this clipping boundary.
III. Another possibility is that the first endpoint is inside the clipping boundary and the second endpoint is outside.
IV. And, finally, both endpoints could be outside the clipping boundary.

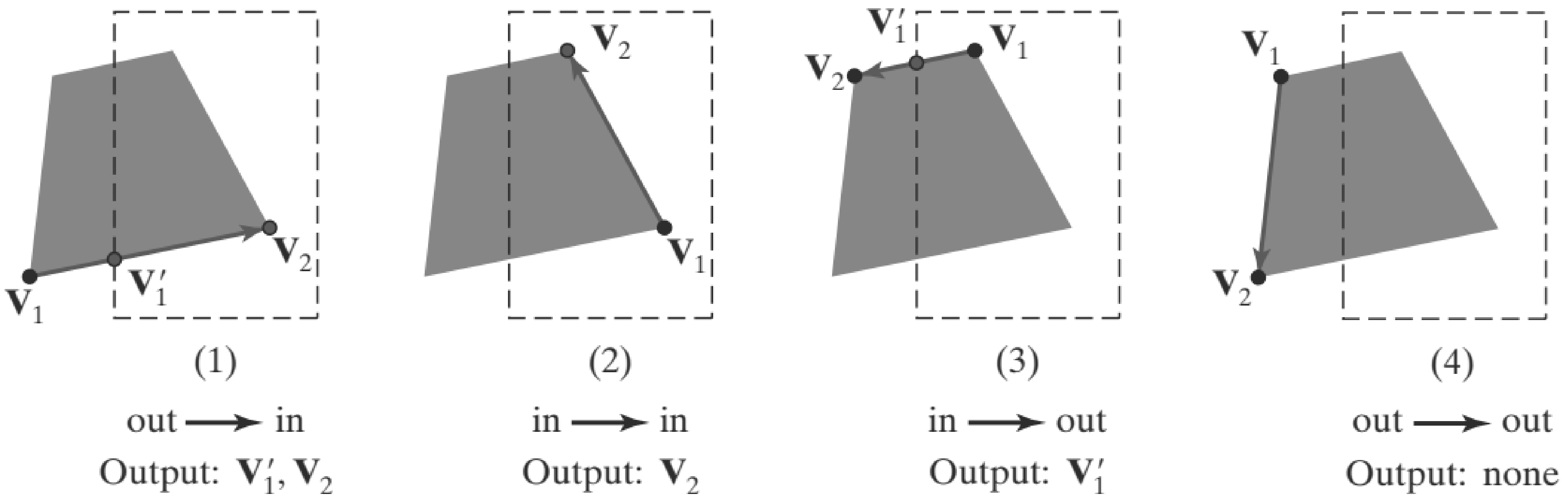


FIGURE 24

The four possible outputs generated by the left clipper, depending on the position of a pair of endpoints relative to the left boundary of the clipping window.

1. If the first input vertex is outside this clipping-window border and the second vertex is inside, both the intersection point of the polygon edge with the window border and the second vertex are sent to the next clipper.

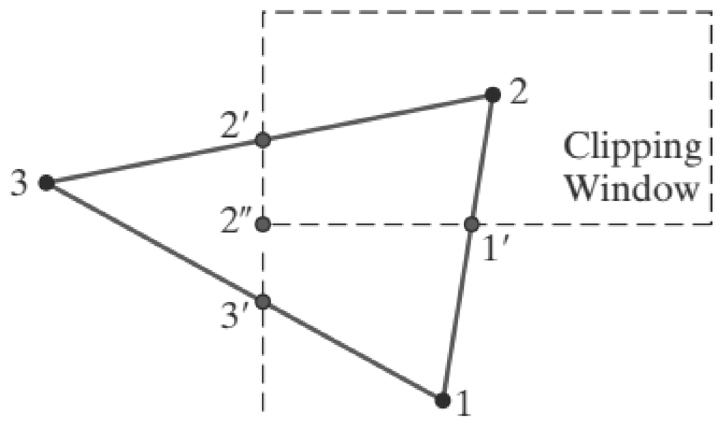
2. If both input vertices are inside this clipping-window border, only the second vertex is sent to the next clipper.

3. If the first vertex is inside this clipping-window border and the second vertex is outside, only the polygon edge-intersection position with the clipping-window border is sent to the next clipper.

4. If both input vertices are outside this clipping-window border, no vertices are sent to the next clipper.

FIGURE 25

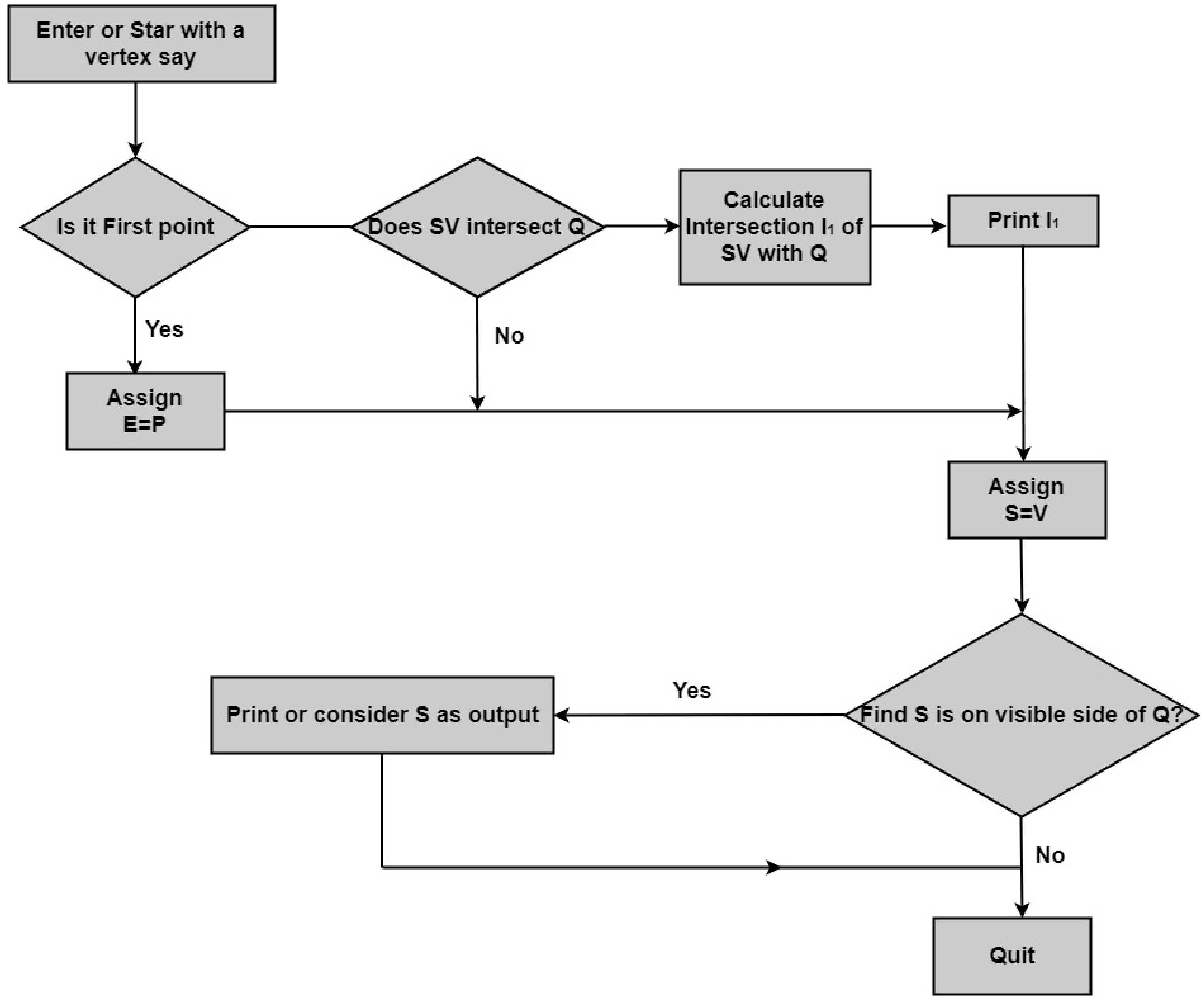
Processing a set of polygon vertices, {1, 2, 3}, through the boundary clippers using the Sutherland-Hodgman algorithm. The final set of clipped vertices is {1′, 2, 2′, 2″}.



$$\begin{tikzcd}
\text{Input Edge:} \arrow[r] & \boxed{\text{Left Clipper}} \arrow[r] & \boxed{\text{Right Clipper}} \arrow[r] & \boxed{\text{Bottom Clipper}} \arrow[r] & \boxed{\text{Top Clipper}}
\end{tikzcd}$$

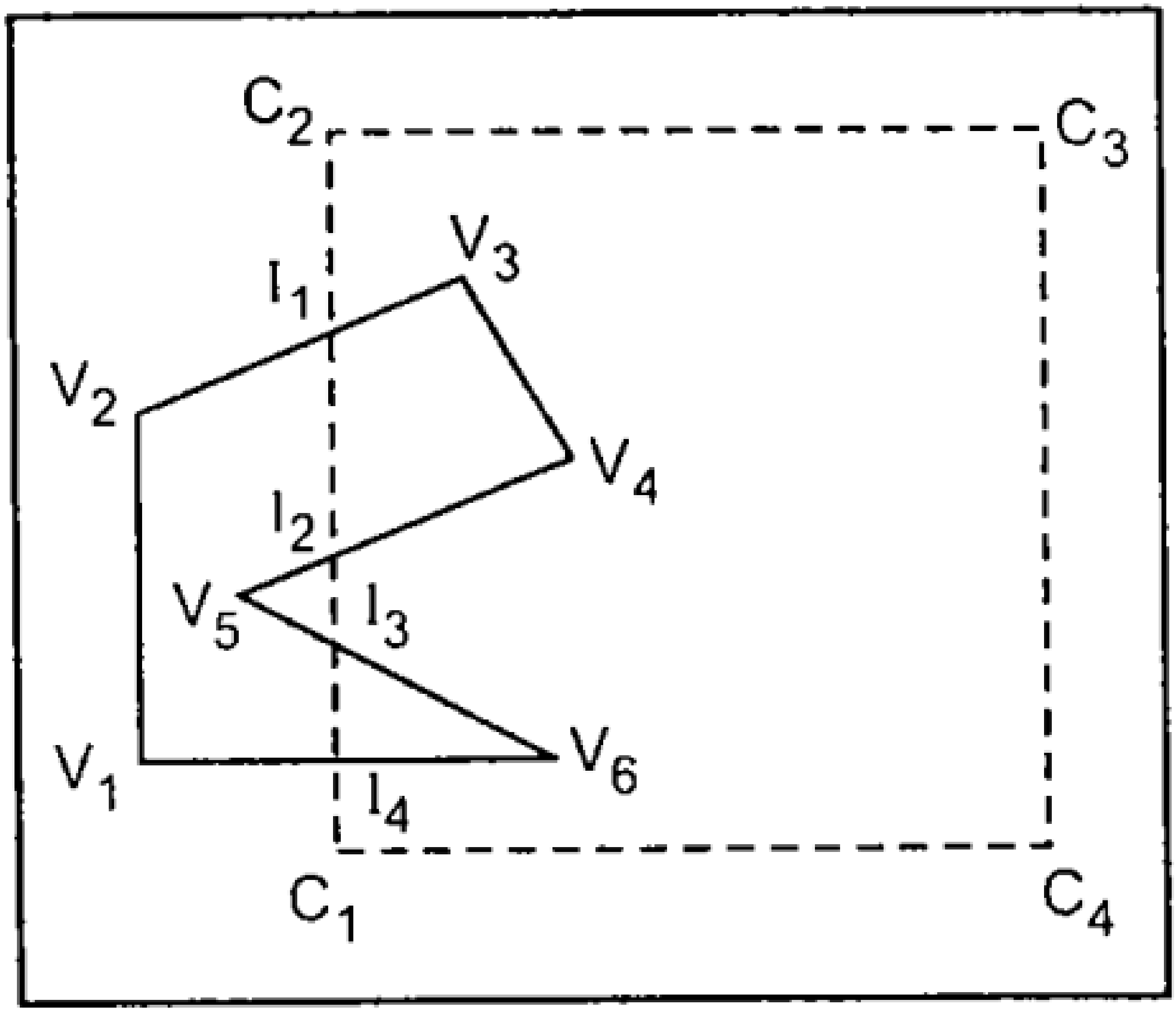
Input Edge:	Left Clipper	Right Clipper	Bottom Clipper	Top Clipper
{1, 2}:	(in – in) → {2}			
{2, 3}:	(in – out) → {2′}	{2, 2′}: (in – in) → {2′}		
{3, 1}:	(out – in) → {3′, 1}	{2′, 3′}: (in – in) → {3′}	{2′, 3′}: (in – out) → {2″}	
		{3′, 1}: (in – in) → {1}	{3′, 1}: (out – out) → { }	
		{1, 2}: (in – in) → {2}	{1, 2}: (out – in) → {1′, 2}	{2″, 1′}: (in – in) → {1′}
			{2, 2′}: (in – in) → {2′}	{1′, 2}: (in – in) → {2}
				{2, 2′}: (in – in) → {2′}
				{2′, 2″}: (in – in) → {2″}

Sutherland Hodgemen Algorithm:


Enter or Star with a vertex say
Is it First point
Yes
Assign E=P
Does SV intersect Q
No
Calculate Intersection I_1 of SV with Q
Print I_1
Assign S=V
Find S is on visible side of Q?
Yes
Print or consider S as output
No
Quit

2. Polygon Clipping: **Weiler-Atherton**

- This algorithm provides a general polygon-clipping approach that can be used to clip a fill area that is either a convex polygon or a concave polygon.
- The method was developed as a means for identifying visible surfaces in the three-dimensional scene.
- Also use this approach to clip any polygon fill area against a clipping window with any polygon shape.
- This algorithm traces around the perimeter of the fill polygon searching for the borders that enclose a clipped fill region.



C_2
C_3
V_3
I_1
V_2
V_4
I_2
V_5
I_3
V_1
V_6
I_4
C_1
C_4

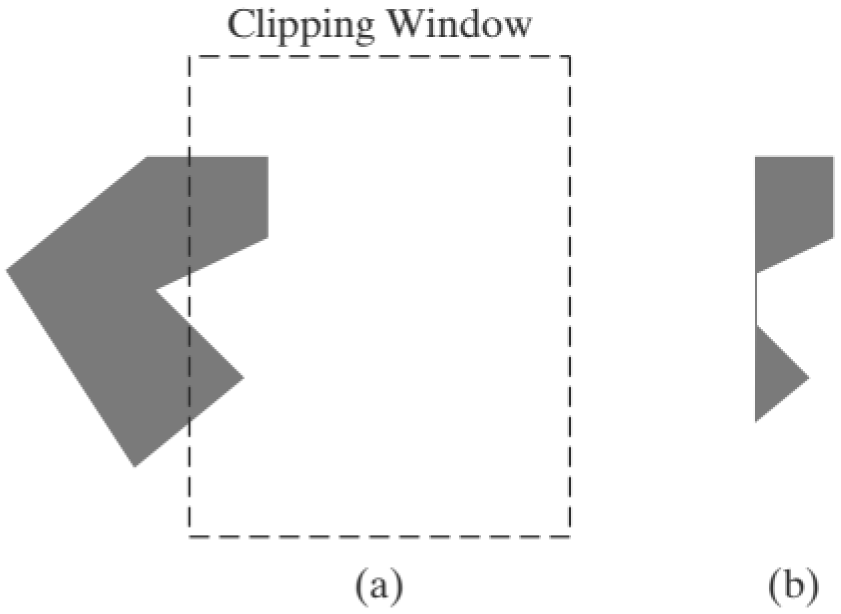


FIGURE 26
Clipping the concave polygon in (a) using the Sutherland-Hodgman algorithm produces the two connected areas in (b).

- To find the edges for a clipped fill area, we follow a path (either counterclockwise or clockwise) around the fill area that detours along a clipping-window boundary whenever a polygon edge crosses to the outside of that boundary.

- The direction of a detour at a clipping-window border is the same as the processing direction for the polygon edges.

- We can determine whether the processing direction is counterclockwise or clockwise from the ordering of the vertex list that defines a polygon fill area.

- In most cases, the vertex list is specified in a counterclockwise order as a means for defining the front face of the polygon.

- Thus, the cross-product of two successive edge vectors that form a convex angle determines the direction for the normal vector, which is in the direction from the back face to the front face of the polygon.

- If we do not know the vertex ordering, we could calculate the normal vector, or we can locate the interior of the fill area from any reference position.

- Then, if we sequentially process the edges so that the polygon interior is always on our left, we obtain a counterclockwise traversal.

- Otherwise, with the interior to our right, we have a clockwise traversal.

For a counterclockwise traversal of the polygon fill-area vertices, we apply the following Weiler-Atherton procedure:

- Process the edges of the polygon fill area in a counterclockwise order until an inside-outside pair of vertices is encountered for one of the clipping boundaries; that is, the first vertex of the polygon edge is inside the clip region and the second vertex is outside the clip region.

- Follow the window boundaries in a counterclockwise direction from the exit-intersection point to another intersection point with the polygon. If this is a previously processed point, proceed to the next step. If this is a new intersection point, continue processing polygon edges in a counterclockwise order until a previously processed vertex is encountered.

- Form the vertex list for this section of the clipped fill area.

- Return to the exit-intersection point and continue processing the polygon edges in a counterclockwise order.

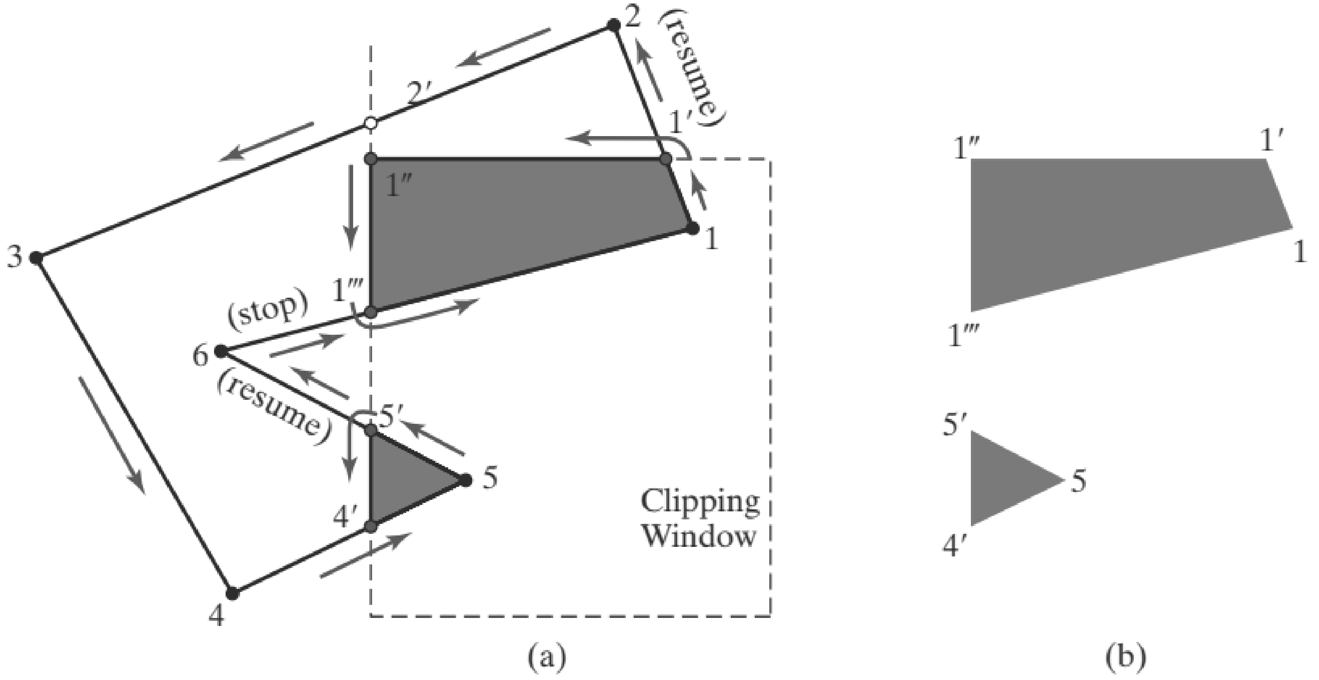


FIGURE 27
A concave polygon (a), defined with the vertex list {1, 2, 3, 4, 5, 6}, is clipped using the Weiler-Atherton algorithm to generate the two lists {1, 1′, 1″, 1‴} and {4′, 5, 5′}, which represent the separate polygon fill areas shown in (b).

- For a clockwise edge traversal, we would use a clockwise clipping-window traversal.
- Starting from the vertex labeled 1, the next polygon vertex to process in a counterclockwise order is labeled 2.
- Thus, this edge exits the clipping window at the top boundary.
- We calculate this intersection position (point 1’) and make a left turn there to process the window borders in a counterclockwise direction.
- Proceeding along the top border of the clipping window, we do not intersect a polygon edge before reaching the left window boundary.
- Therefore, we label this position as vertex 1’’ and follow the left boundary to the intersection position 1’’’.

- We then follow this polygon edge in a counterclockwise direction, which returns us to vertex 1.
- This completes a circuit of the window boundaries and identifies the verted list {1, 1’, 1’’, 1’’’} as a clipped region of the original fill area.
- Processing of the polygon edges is then resumed at point 1’.
- The edge defined by points 2 and 3 crosses to the outside of the left boundary, but points 2 and 2’ are above the top clipping-window border and point 2’ and 3 are to the left of the clipping region.

- Also, the edge with endpoints 3 and 4 is outside the left clipping boundary, but the next edge (from endpoint 4 to endpoint 5) re-enters the clipping region and we pick up intersection point 4’.

- The edge with endpoints 5 and 6 exits the window at intersection position 5’, so we detour the left clipping boundary to obtain the closed vertex list {4’, 5, 5’}.

- We resume the polygon edge processing at position 5’, which returns us to the previously processed point 1’’’.

- At this point, all polygon vertices and edges have been processed, so the fill area is completely clipped.

IV. Curve Clipping

- We can first test the coordinate extents of an object against the clipping boundaries to determine whether it is possible to accept or reject the entire object trivially.
- If not, we could check for object symmetries that we might be able to exploit in the initial accept/reject tests.
- For example, circles have symmetries between quadrants and octants.

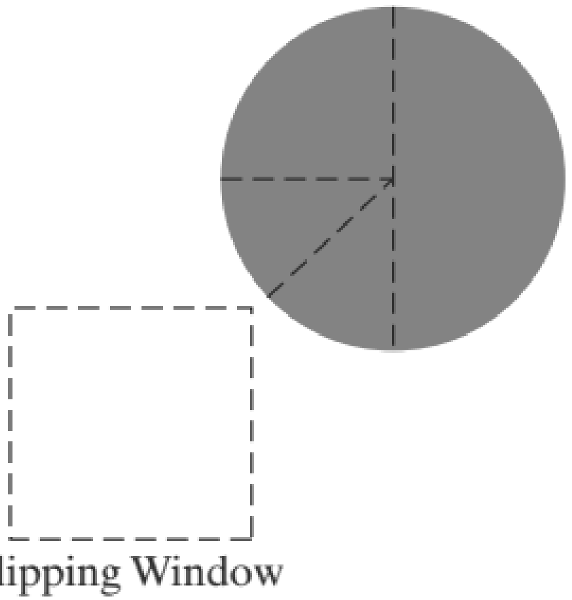


FIGURE 29
A circle fill area, showing the quadrant and octant sections that are outside the clipping-window boundaries.

- An intersection calculation involves substituting a clipping-boundary position (xw_{min}, xw_{max}, yw_{min}, or yw_{max}) in the nonlinear equation for the object boundary and solving for the other coordinate value.
- Once all intersection positions have been evaluated, the defining positions for the object can be stored for later use by the scan-line fill procedures.
- The circle radius and the endpoints of the clipped arc can be used to fill the clipped region, by invoking the circle algorithm to locate positions along the arc between the intersection endpoints.

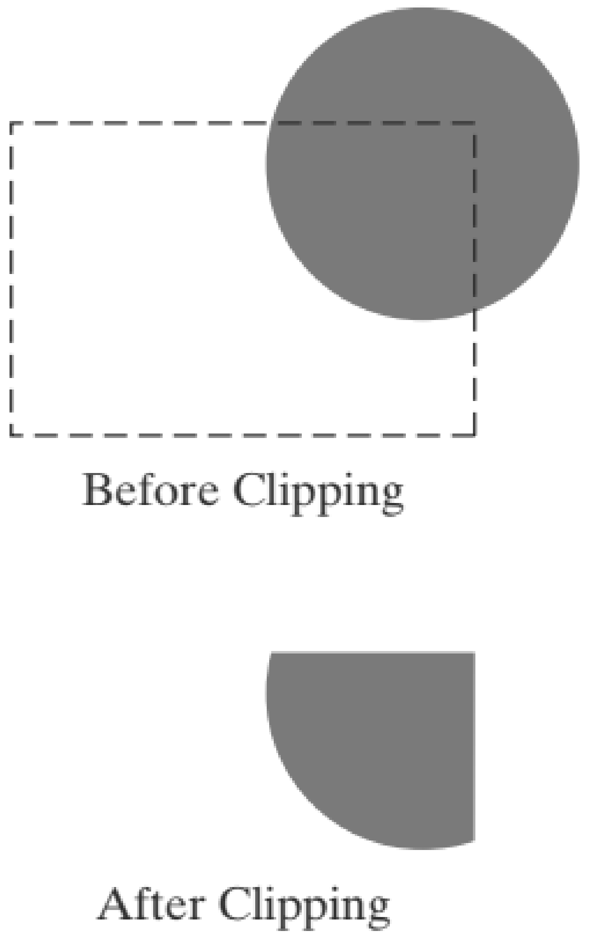


FIGURE 30
Clipping a circle fill area.

- Similar procedures can be applied when clipping a curved object against a general polygon clipping region.

- On the first pass, we could compare the bounding rectangle of the object with the bounding rectangle of the clipping region.

- If this does not save or eliminate the entire object, we next solve the simultaneous line-curve equations to determine the clipping intersection points.

V. Text Clipping

- The simplest method for processing character strings relative to the limits of a clipping window is to use the *all-or-none string-clipping* strategy.
- If all of the string is inside the clipping window, we display the entire string. Otherwise, the entire string is eliminated.
- This procedure is implemented by examining the coordinate extents of the text string.
- If the coordinate limits of this bounding rectangle are not completely within the clipping window, the string is rejected.

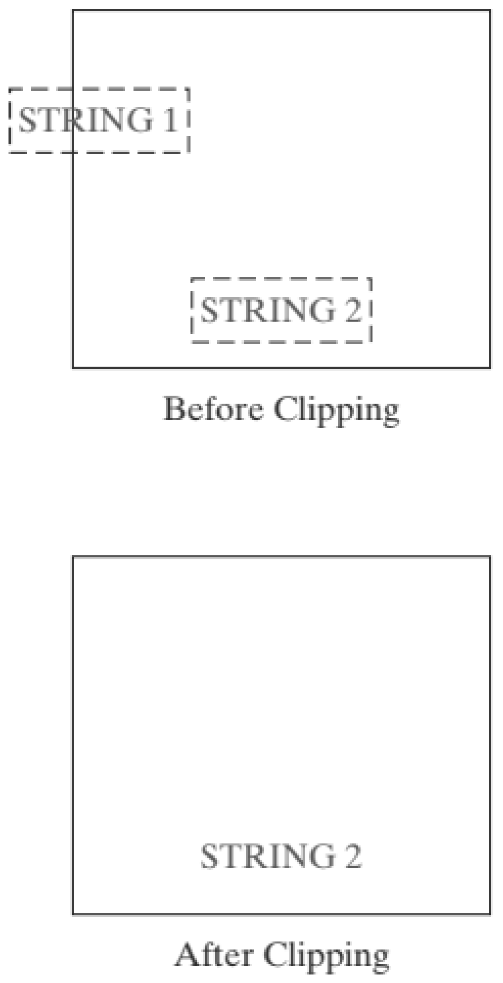


FIGURE 31
Text clipping using the coordinate extents for an entire string.

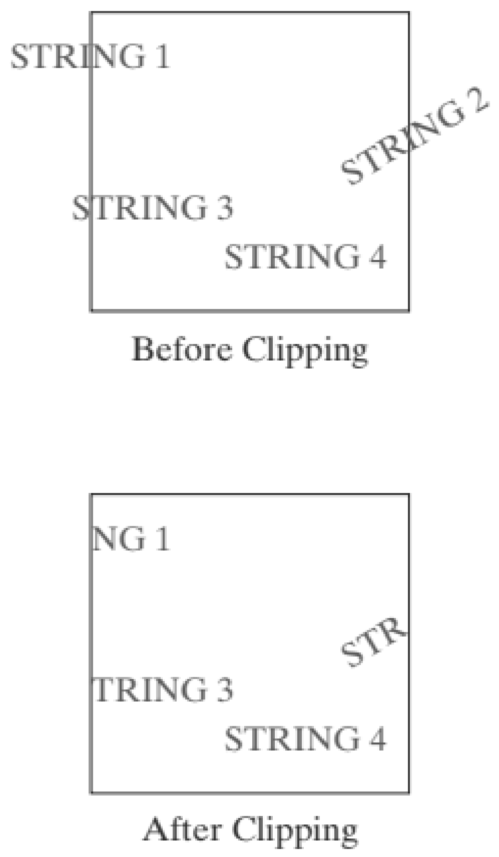


FIGURE 32
Text clipping using the bounding rectangle for individual characters in a string.

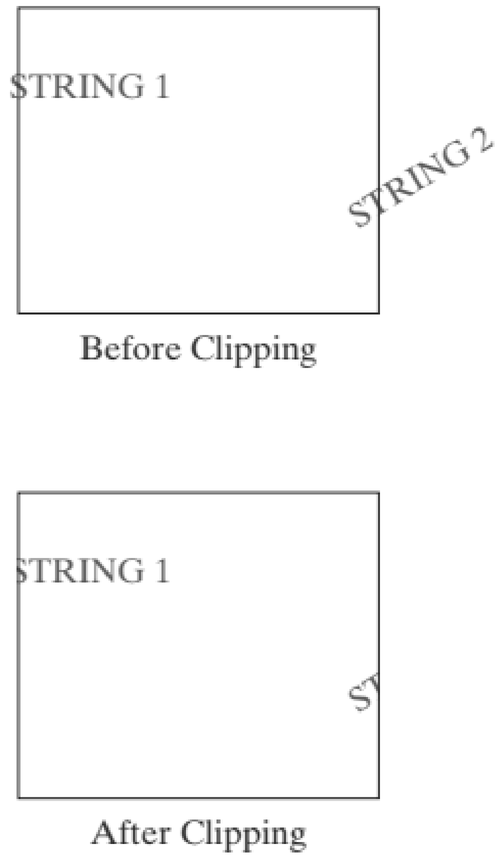


FIGURE 33
Text clipping performed on the components of individual characters.

- An alternative is to use the all-or-none character-clipping strategy.
- Here we eliminate only those characters that are not completely inside the clipping window.
- In this case, the coordinate extents of individual characters are compared to the window boundaries.
- Any character that is not completely within the clipping-window boundary is eliminated.
- A third approach to text clipping is to clip the components of individual characters.
- This provides the most accurate display of clipped character strings, but it requires the most processing.

- If an individual character overlaps a clipping window, we clip off only the parts of the character that are outside the window.

- Outline character fonts defined with line segments are processed in this way using a polygon-clipping algorithm.

- Characters defined with bit maps are clipped by comparing the relative position of the individual pixels in the character grid patterns to the borders of the clipping region.